(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
10 May 2001 (10.05.2001)

PCT

(10) International Publication Number
WO 01/33387 A2

(51) International Patent Classification[7]: G06F 17/00

(21) International Application Number: PCT/CA00/01280

(22) International Filing Date: 27 October 2000 (27.10.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

60/162,717	29 October 1999 (29.10.1999)	US
60/169,454	7 December 1999 (07.12.1999)	US
60/169,455	7 December 1999 (07.12.1999)	US
60/179,595	1 February 2000 (01.02.2000)	US
60/198,396	19 April 2000 (19.04.2000)	US

(71) Applicant *(for all designated States except US)*: **LIBERTY INTEGRATION SOFTWARE, INC.** [CA/CA]; Suite 126, 1020 Mainland Street, Vancouver, British Columbia V6B 5L1 (CA).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **HOUBEN, Robert** [CA/CA]; Suite 126, 1020 Mainland Street, Vancouver, British Columbia V6B 5L1 (CA). **HUNTER, John** [CA/CA]; Suite 126, 1020 Mainland Street, Vancouver, British Columbia V6B 5L1 (CA). **MANSFIELD, Philip** [CA/CA]; Suite 126, 1020 Mainland Street, Vancouver, British Columbia V6B 5L1 (CA). **KHRAMOV, Yuri** [CA/CA]; Suite 126, 1020 Mainland Street, Vancouver, British Columbia V6B 5L1 (CA).

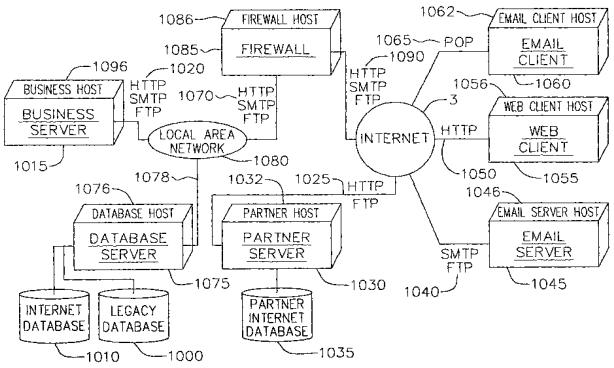
(74) Agents: **CLARK, Neil, S.** et al.; Fetherstonhaugh & Co., Box 11560, Vancouver Centre, Suite 2200, 650 West Georgia Street, Vancouver, British Columbia V6B 4N8 (CA).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

[Continued on next page]

(54) Title: APPARATUS, SYSTEMS AND METHODS FOR ELECTRONIC DATA DEVELOPMENT, MANAGEMENT, CONTROL AND INTEGRATION IN A GLOBAL COMMUNICATIONS NETWORK ENVIRONMENT



(57) Abstract: Apparatus, systems and methods are provided for integrating data stored using legacy Data Base Management Systems (sometimes referred to herein as "legacy data") with data that is accessible through a global communications network environment (sometimes referred to herein as "Internet data") such as the Internet. A server provides a structured process for efficient data flow through incoming document transformation, implementation of business rules, and response document routing. Legacy and Internet data are converted and integrated, without loss, using intermediate data structures encapsulated within software objects with exposed methods for creation, navigation, maintenance, and accessing of data within the software objects. These software objects may also be used for developing, managing, controlling and integrating Internet data from within any Windows scripting hosted language, including among others, VB (Visual Basic) Script, JScript, and PerlScript. In an alternative embodiment, a legacy data Internet portal is provided for legacy application software systems integration with a global communications network such as the Internet that exposes the ability to execute a method on a legacy database server using an Internet application on a Web server. The results of the executed method are returned to the Web server as a tree-based Internet data structure.

WO 01/33387 A2

Published:

— *Without international search report and to be republished upon receipt of that report.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

APPARATUS, SYSTEMS AND METHODS FOR ELECTRONIC DATA DEVELOPMENT, MANAGEMENT, CONTROL AND INTEGRATION IN A GLOBAL COMMUNICATIONS NETWORK ENVIRONMENT

BACKGROUND OF THE INVENTION

Pre-Internet era businesses are struggling to integrate their businesses and existing legacy systems with the global communications network known as the Internet. (Many businesses originated in the pre-Internet era and automated their respective data processing needs prior to the development and wide-spread acceptance of the Internet. These pre-Internet systems are often referred to as legacy systems. Other entities have businesses that revolve primarily around the Internet; typically, such businesses originated during or after the dawning of the Internet age and are sometimes referred to herein as "Internet-era businesses"). Internet-era businesses are struggling with unrefined tools to develop, manage and control data collected and processed with their Internet-based systems. Both Pre-Internet era and Internet-era businesses are struggling to communicate with one another.

In some cases, the data structures used by legacy systems are not directly compatible with Internet data structures. Consequently, many enterprises with legacy systems face the task of converting data created and stored by their respective legacy systems to a form cognizable by Internet based systems for their respective emergence into electronic commerce (eCommerce), and for their interface with their respective Internet trading partners.

Additionally, eCommerce may comprise only a portion of an enterprise's entire business. Accordingly, data collected and processed on the Internet must be integrated with the data maintained on the enterprise's legacy systems.

Most automated systems, whether legacy or Internet, use some architectural scheme, which is expressed in a particular data structure, to organize the system's data. There are many different types of data structures. Legacy systems frequently use hierarchical or multivalued simple data structures with which to organize their respective data. Some legacy systems use relational simple data database management systems. In contrast to the data structures used by legacy systems, tree-based rich data structures comprise much of the data foundation for the Internet.

In contrast to the explicit organizational and descriptive intelligence contained in rich data structures, the organization and meaning of data items in simple data structures are often inherent to the order of the data, and/or require the intelligence of what is known as metadata or an application program to identify the beginning and end of, and impart meaning to, individual data items.

Metadata is data about data -- it is high level data about the lower-level data contained in a particular file or data base. In some cases, such as is the case with multivalued data structures, low level metadata is implicit in the data which it describes. In other cases, such as is the case

with certain Data Base Management Systems (DBMS), metadata describing the data base is contained within a data dictionary. Other examples of metadata include: record descriptions in a COBOL program, CASE entity relationship diagrams for a particular set of entities, and data server catalog logical view descriptions.

As a consequence of the absence of explicit metadata in simple data structures, conversion to tree-based rich data structures based on the source data alone may result in data loss.

In a tree-based rich data structure, a root node, also referred to as a mother or parent node, describes the most basic level of information about the data to which the root node pertains. For instance, a document level node is used to describe a document. Other nodes, referred to as "child" or "children" nodes, can be designated with some relation, either direct or indirect, to the root node. For instance, the root node may have one or more child nodes, each of which in turn has one or more child nodes, each of which in turn has multiple children nodes.

One of the predominant tools for the development and exchange of data in Internet-based systems is Extensible Markup Language (XML). XML was originally designed as a markup language for electronic documents. A mark up language such as XML uses certain defined delimiters and tag names to designate meaning and/or organization of marked text within an electronic document. As an example, a sample electronic document has as its title the words "This is the Title" and has as a single paragraph of text "This is the text." Using an exemplary mark up language to mark up the electronic document, a start title delimiter/tag name (in this example, "<t>") is inserted before the title text for an electronic document; an end title delimiter/tag name (in this example, "</t>") is inserted after the title text. Similarly, a start paragraph delimiter/tag name (in this example, " ") is inserted before the first letter of the first word of a particular paragraph; an end paragraph delimiter/tag name (in this example, " ") is inserted after the last letter of the last word of the paragraph. The resulting marked up electronic document would be represented in memory as follows:

<t> This is the Title </t>
 This is the text.

Internet development programmers have begun to use document mark up languages, such as XML, to develop and exchange many types of data collections, other than merely electronic documents (electronic documents are sometimes referred to herein as simply "documents"). XML is used to identify structures in such diverse applications as metadata description, vector graphic manipulation, eCommerce, and mathematical equation expression, to name just a few.

In addition to being a mark up language, XML is also what is known as a "meta language" in that it provides for the explicit declaration of new Document Type Definitions (DTD) -- that is, it provides development programmers with the ability to define program-specific tag names.

Because of the DTD declaration capability of XML, each business may independently develop its own XML structures and tags. A particular strategy for marking up a document with tag names, naming conventions, delimiters and/or document structures is referred to herein as a "mark up schema" or simply "schema".

An electronic document, or other collection of data (data collection), that has been marked up, such as with XML delimiters and tag names, is referred to as a marked up document; in the case of XML, as an XML document. Application programs are typically designed to access, navigate and manipulate marked up documents in tree-based form. Application programs access XML documents through a Document Object Model (DOM). A DOM is a machine-readable tree-based representation of an XML document. A type of program sometimes referred to as a document parser (in the case of XML, as an XML parser) provides a translation interface between the marked up document and the machine readable tree-based representation of that marked up electronic document.

The structural elements and DTD capabilities of XML and DOM enable the development of tree-based rich data structures. Notwithstanding the fact that many Internet developers are using XML and other mark up languages to define data structures for Internet applications, the tools, such as DOM, that are available to create, navigate and maintain tree-based rich data structures are cumbersome and difficult to use.

Without specially designed technology, the job of converting data without data loss from legacy systems to Internet-cognizable tree-based data structures can be a complex, resource-intensive project for any company tackling the job; the job is one that, with existing tools, requires highly-detailed, fact-specific program coding. Furthermore, without better, more effective tree-based rich data structure navigational tools, the creation, navigation, maintenance, and accessing of data between multiple Internet enterprises will remain difficult and time consuming.

Technology is therefore required to facilitate the resource efficient conversion of data created by legacy systems to a form useful to Internet-based programs and to facilitate the integration of the converted data with data existing in the Internet environment. Technology is further required to facilitate the resource efficient conversion of Internet data to simple data structures with which legacy systems can communicate and to facilitate the integration of the converted data with existing legacy system data. Technology is still further required to facilitate the resource efficient creation, navigation, maintenance, and accessing of data between multiple Internet enterprises.

Another aspect of Legacy-Internet transitions is the investment many companies have made in their respective legacy systems. Companies that have made such an investment and which also want to gain the benefits of the latest Internet technology, marketing and business opportunities, need a way to leverage the existing legacy system application software. Accordingly, some means is required that can expose (the term "expose" is used herein to mean

"make available for access", "make accessible" and/or "access") the software capability and functionality of legacy system application software.

SUMMARY OF THE INVENTION

The present invention provides apparatus, systems and methods for integrating data stored using legacy Data Base Management Systems (sometimes referred to herein as "legacy data") with data that is accessible through a global communications network environment (sometimes referred to herein as "Internet data") such as the Internet.

One aspect of the present invention provides apparatus, systems and methods for converting and integrating, without data loss, legacy data to an Internet data structure, such as a tree-based rich data structure. In one embodiment, the present invention provides a method, using a computer, for converting source data to an Internet data structure. The Internet data structure is characterizable by a tree-based structure definition. The source data to be converted is characterized by a particular structure and by a set of relationships. The structure and the set of relationships that characterize the source data are described by metadata which comprises a plurality of metadata components. The metadata and the metadata components are accessible by the computer.

The method to convert the source data to the Internet data structure comprises several steps. The computer translates the metadata into a set of DTD declarations and a corresponding tree-based structure definition. If the metadata is not in a form readily accessible by the computer, then the first step is to translate the metadata into a form that is readily accessible by the computer.

The computer develops a conversion map comprised of equivalent DTD declarations and the corresponding metadata component and further comprised of equivalent node definition and the corresponding metadata component. The computer then uses the conversion map to map the source data into the Internet data structure.

Yet another aspect of the present invention provides apparatus, systems and methods for converting and integrating, without data loss, Internet data to a legacy data structure. In one embodiment, the present invention provides a method, using a computer, for converting Internet data to a simple legacy data structure. The Internet data structure is characterized by a tree-based structure definition. The legacy data structure is characterized by a particular structure and by a set of relationships. The structure and the set of relationships that characterize the legacy data are described by metadata which comprises a plurality of metadata components. The metadata and the metadata components are accessible by the computer.

The method to convert the Internet data to the legacy data structure comprises several steps. The computer translates the metadata into a set of DTD declarations and a corresponding receiving tree-based structure definition. The receiving tree-based structure definition comprises a plurality of receiving node definitions. Each DTD declaration is equivalent

to one of the metadata components. Each of the receiving node definitions is equivalent to one of the metadata components.

The computer stores each equivalent DTD declaration and the corresponding metadata component as a conversion map components. The computer stores each equivalent receiving node definition and the corresponding metadata component as a conversion map component. The computer then aggregates the conversion map components into a conversion map. The computer uses the conversion map to create a receiving tree-based structure.

The computer then compares the Internet data tree-based structure with the receiving tree-based structure. For each node in the Internet data tree-based structure, the computer identifies the equivalent receiving tree-based structure node. The computer builds a translation table of the equivalent Internet data tree nodes and the corresponding receiving tree nodes. The computer uses the translation table to map the Internet data to the receiving tree-based structure. Then, using the conversion map, the computer translates the data in the receiving tree-based structure into the legacy data structure.

Yet another aspect of the present invention provides apparatus, systems and methods for the resource efficient creation, navigation, maintenance, and accessing of data between multiple Internet enterprises. In one embodiment, the present invention provides a method, using a computer, for converting Internet data from one Internet data structure to another Internet data structure. The first Internet data structure is characterized by a tree-based structure definition. The second Internet data structure is characterized by a second tree-based structure definition.

The method for converting Internet data from one Internet data structure to another Internet data structure comprises several steps. The first tree-based structure definition comprises a first plurality of node definitions. The second tree-based structure comprises a second plurality of nodes.

The computer compares the first Internet data tree-based structure with the second tree-based structure. For each node in the first Internet data tree-based structure, the computer identifies the equivalent node in the second tree-based structure. The computer builds a translation table of the equivalent first Internet data tree nodes and the corresponding second Internet data tree nodes. The computer uses the translation table to map the first Internet data to the second Internet data tree-based structure.

Still another aspect of the present invention provides apparatus, systems and methods for developing, managing, controlling and integrating Internet data from within any Windows Scripting Hosted language, including among others, VB (Visual Basic) Script, JavaScript, PerlScript, and others using a simplified dot nodepath notation that provides for dynamic changes, additions and deletions of data nodes without the need for intervening compilation steps.

A further aspect of the invention is an Electronic Portal for Legacy Line-of-Business application software systems integration with a global communications network such as the

Internet. This Electronic Portal exposes the ability to execute a command on a Legacy Line-of-Business database server using an Internet application on a Web server. The results of the executed command are returned to the Web server as an Internet data structure.

BRIEF DESCRIPTION OF THE DRAWINGS

These and other features, aspects, and advantages of the present invention will become better understood with regard to the following description, appended claims, and accompanying drawings where:

FIG. 1a is a deployment diagram of and exemplary deployment across multiple host computers of an exemplary Internet based system for accessing legacy data;

FIG. 1b is a high level diagram of the use of a legacy data to Internet data adapter used to access a single legacy database;

FIG. 2 is a diagram of the architecture of an exemplary general purpose computer capable of serving as a host for the software objects depicted in FIG. 1a;

FIG. 3 is a sequence diagram of an exemplary communication sequence between the software objects of FIG. 1;

FIG. 4 depicts data flow diagrams for exemplary legacy data adapter and Internet data translator objects;

FIG. 5 is a depiction of an exemplary DOM object encapsulating a DOM and exposing methods for operations on the DOM;

FIG. 6a is a class diagram for a business server;

FIG. 6b is a data flow diagram illustrating intake of documents from the Web by the business server;

FIG. 6c is a dataflow diagram of data through a business server;

FIG. 6d is a data flow diagram of the operations of a sorter object according to the present invention;

FIG. 6e is an exemplary business server XML document;

FIG. 6f is an exemplary pipeline XML document as used by a dispatcher;

FIG. 7 is a cooperation diagram depicting the operations in business server during an exemplary business transaction;

FIG. 8 is a graphic representation depicting the structural element wrapping architectural feature of the invention;

FIG. 9 provides a simplified, high level flow diagram depicting an exemplary embodiment of exemplary legacy data to Internet data conversion program logic flow;

FIGS. 10 and 11 depict typical invoice data represented in a hierarchical manner;

FIG. 12 depicts an exemplary XML structure of typical invoice data;

FIG. 13 is a formatted representation of an invoice record in an exemplary legacy data structure;

FIG. 14 is a representation of the way in which the record depicted in FIG. 13 is stored in memory;

FIGS. 15 and 16 are a representation of the equivalent Internet data structure of the data record depicted in FIG. 14;

FIG. 17 depicts exemplary Visual Basic code that executes an exemplary syntax of a LoadDOM method in an exemplary embodiment of the invention;

FIG. 18 depicts exemplary Visual Basic code that executes an exemplary syntax of a validate method in an exemplary embodiment of the invention;

FIGS. 19-22 depict exemplary XML fragments;

FIG. 23 is a graphical representation of an exemplary tree-based rich data structure in an exemplary Internet data structure.

FIG. 24 is a logic flow diagram depicting the functional logic of an exemplary embodiment of an ExtendedProperties method;

FIG. 25 represents an XML document containing exemplary data;

FIG. 26 depicts exemplary scripting language program code using the dot nodepath notation aspect of the invention;

FIG. 27 is a logic flow diagram depicting the functional logic of an exemplary embodiment of the Node collection retrieval feature of the invention;

FIG. 28 depicts exemplary Visual Basic code that executes an exemplary syntax of an exemplary embodiment of the Node collection retrieval feature of the invention;

FIG. 29 is a logic flow diagram depicting the functional logic of an exemplary embodiment of the Node namepath interpretation feature of the invention;

FIG. 30 is a block diagram depicting the relationships between a three-tier system architecture using the Electronic Portal aspect of the invention;

FIG. 31 depicts an exemplary Legacy method and an exemplary response in an exemplary Legacy Line-of-Business software application system;

FIG. 32 depicts an exemplary Legacy data Internet portal method using XML;

FIG. 33 depicts the output of an exemplary Legacy data Internet portal method;

FIG. 34 is a logic flow diagram depicting the functional logic of an exemplary embodiment of the Create a Q-Pointer feature of the invention;

FIGS. 35 and 36 are graphic representations of interactive online display graphic user Q-Pointer interface features of the invention;

FIGS. 37 and 38 are graphic representations of interactive online display graphic user Q-Pointer interface features of the invention;

FIG. 39 is a logic flow diagram depicting the functional logic of an exemplary embodiment of the GetFile feature of the invention;

FIG. 40 is a graphic representation of an interactive online display graphic user GetFile interface feature of the invention;

FIG. 41 is a graphic representation of a URL resulting from user entry of GetFile specification information;

FIG. 42 depicts exemplary code embodying Retrieval Logic functions involved in Specifying the Nature of a GetFile Request;

FIG. 43 is a graphic representation of an interactive online display graphic user GetFile specification interface feature of the invention;

FIG. 44 depicts exemplary code embodying Retrieval Logic functions for Retrieving GetFile Data;

FIG. 45 is a graphic representation of an interactive online display graphic user GetItem interface feature of the invention;

FIG. 46 is a graphic data hierarchy diagram depicting an exemplary representation of an exemplary Relational Database Management System configuration.

DETAILED DESCRIPTION OF THE INVENTION

Server Architecture

FIG. 1 is a deployment diagram of an exemplary implementation of a system facilitating the use of legacy data within Internet based transactions. The deployment diagram depicts a network of software objects with each software object deployed on a host. Each host is a general purpose computer as depicted in FIG. 2.

FIG. 2 is a hardware architecture diagram of a general purpose computer suitable for use as a software object host. Microprocessor 3600, comprised of a Central Processing Unit (CPU) 3610, memory cache 3620, and bus interface 3630, is operatively coupled via system bus 3635 to main memory 3640 and I/O control unit 3645. The I/O interface control unit is operatively coupled via I/O local bus 3650 to disk storage controller 3695, video controller 3690, keyboard controller 3685, and communications device 3680. The communications device is adapted to allow software objects hosted by the general purpose computer to communicate via a network with other software objects. The disk storage controller is operatively coupled to disk storage device 3625. The video controller is operatively coupled to video monitor 3660. The keyboard controller is operatively coupled to keyboard 3665. The network controller is operatively coupled to communications device 3696.

Computer program instructions implementing a software object are stored on the disk storage device until the microprocessor retrieves the computer program instructions and stores them in the main memory. The microprocessor then executes the computer program instructions stored in the main memory to implement the software object.

Alternatively, other types of general purpose computers are used to host a business server. For example, the business server may be hosted by Personal Digital Assistants (PDAs) or computer systems dedicated to hosting Internet servers.

Referring again to FIG. 1, business host 1096 hosts business server 1015 and is

operatively coupled to local area network 1080 via business communications link 1020. The business communications link is adapted to support various communications protocols based on the Transport Control Protocol/Internet Protocol (TCP/IP) such as Hyper Text Transfer Protocol (HTTP), Simple Mail Transfer Protocol (SMTP) and File Transfer Protocol (FTP) among others. The business server uses the software services and hardware links provided by the business host to communicate with other software objects across the local area network and the Internet. Database host 1076 hosts database server 1075 and is operatively coupled to the local area network via database link 1078. The database server provides access to Internet database 1010 and legacy database 1000. The business server stores and retrieves data from the Internet database and the legacy database using services provided by the database server. The legacy database contains data accessible in a format not normally suited for Internet applications. The business server provides services to Internet based clients to retrieve, manipulate, transmit, and store legacy data using the database server. Other databases accessible to the business server, such as the Internet database served by the database server, contain documents suitable for transfer over the Internet using one of the suite of Internet communications protocols such as HTTP or SMTP. The business server and the database server communicate to each other via the local area network.

Firewall host 1086 hosts firewall 1085. The firewall host is operatively coupled to the local area network via internal firewall communications link 1070. The internal firewall communications link is adapted to support various communications protocols based on TCP/IP such as HTTP, SMTP, and FTP, among others. The firewall host is operatively coupled to Internet 3 via external firewall communications link 1090. The external firewall communications link is adapted to support various communications protocols based on TCP/IP such as HTTP, SMTP, and FTP, among others. The Internet is commonly called the World Wide Web (WWW or Web) when software objects communicate to each other over the Internet using one of a suite of communications protocols based on TCP/IP such as HTTP. In alternative embodiments, the Internet may be replaced using any computer network system capable of supporting electronic document transfers such as a local area network or a virtual proprietary network.

The firewall filters incoming Internet data packets to ensure that only data packets for specified communications protocols are passed from the Internet to the local area network. The business server communicates with software objects over the Internet through the firewall using a variety of communications protocols for communication. Exemplary communications protocols are: HTTP used for the transfer of documents written in one of various mark up languages such as Hyper Text Markup Language (HTML) or XML; Simple Mail Transfer Protocol (SMTP) for the transfer of electronic mail (email) messages; and File Transfer Protocol (FTP) for the transfer of text and binary files.

An exemplary software object capable of sending messages to the business server via the

Internet is exemplary partner server 1030. The partner server is hosted by partner host 1032 that is operatively coupled to the Internet via partner communications link 1025. The partner communications link is adapted to support various communications protocols based on TCP/IP such as HTTP, SMTP, and FTP, among others. In operation, the business server and the partner server request and send data to each other over the Internet. For example, the partner server may send an Internet document composed in XML to the business server. The content and structure of the XML document may be decomposed by the business server to create instructions for a purchase order. This purchase order may require the updating of the legacy database and retrieval of an Internet document from the Internet database. The retrieved Internet document is sent by the business server back to the partner server as an order acknowledgment. Any business transaction involving the transfer of data may be implemented so long as the business server and the partner server agree on the appropriate data structure and communications protocol.

Business server 1015 may also communicate over the Internet to other software objects. The business server may send email messages to email server 1045 hosted by email server host 1046 using SMTP as the communications protocol The email server host is operatively coupled to the Internet via email server communications link 1040. The email server communications link is adapted to support mail communications protocols based on TCP/IP such as SMTP and Post Office Protocol (POP). The email messages are held by the email server until they are retrieved by email client 1060 hosted by email client host 1062. The email client host is operatively coupled to the Internet via email client communications link 1065. The email client communications link is adapted to support various communications protocols based on TCP/IP such as POP. The business server uses email messages to send informational messages about the operations of the business server. For example, the business server may send an email message to the email server to acknowledge a purchase order or to alert a customer that a product is on back order.

The business server may also send Internet documents over the Internet in response to requests from exemplary Web client 1055 using HTTP as the communications protocol. The Web client is hosted by Web client host 1056. The Web client host is operatively coupled to the Internet via Web client communications link 1050. The Web client communications link is adapted to support various communications protocols based on TCP/IP such as HTTP. The Web client is used to obtain information from the business server. For example, the Web client may send a request to the business server for an HTML document containing a product catalog. The business server finds the HTML catalog document and forwards the HTML catalog document Web client. The Web client interprets the HTML catalog document to create a catalog display.

FIG. 1b is a simplified high level graphical representation of an exemplary set of interactions between multiple business systems in an exemplary e-commerce application.

In FIG. 1b, a user's computer 1a is connected via communications lines 2a and 2b to the Internet 3. The user's computer is equipped with memory 1b, a keyboard for user input 1f, a display monitor 1d and other user input devices such as a track ball 1e. Web browser software 1c is installed in the memory 1b of the user's computer 1a.

As the user issues commands, the Web browser software 1c responds by sending instructions to and receiving instructions from the Internet 3 and each Web site 4 to which the user chooses to communicate.

The user accesses a particular Web site 4 using the Web browser 1c. Interacting with the Web browser 1c, the user accesses a particular features on a particular Web page of the Web site 4 which in this case provides the user with an opportunity to interact with data contained in a data base 11 that is accessible under a particular DBMS (Data Base Management System) through the Web server 13.

The user may ask to view, select, adjust the content of, or otherwise access the data presented on the Web page 4. The data presented on the Web page contains, for example, a catalog of goods with a price-list. In response to a user request to access the data base 11, the user's Web browser 1c sends a requests to the Web site for data from the DBMS. In the example, the user requests pricing information about a particular item.

The user's request to access data causes the Web page to activate an Active Server Page ("ASP") 14 resident in the Web server 13. The Active Server Page 14 creates HTML (Hyper Text Markup Language) and script commands that instruct the Web server 13 for the Web site 4 to perform certain processing.

In the example, the Web server 13 instructs the ASP 14 to request 5 pricing source information from the data base 11 in response to the user's request; prepares a pricing request from the information obtained 5 from the data base 11; and sends 7 the pricing request to a trading partner 9a for, in this example, pricing information about the particular catalog item chosen by the user. The server can communicate 7 with any number of trading partners, 9a . . . 9n. In the example, the trading partner 9a responds 7 with the pricing information requested. The ASP 14 then builds HTML commands and sends 2b,2a, them to the browser 1c. The browser 1c displays the information about the transaction with the trading partner and allows the user to interact with this data on the browser in order to refine the user's selections and payment options. The user is typically unaware that the user's browser or the Web site accessed is interacting with anything other than the Web site with which the user originally asked to communicate. In the example, once the user views the price for the item, the user chooses to purchase the selected item and to pay for the purchase with a credit card. A financial institution 10a is contacted 8 by the ASP 14 to authorize 8 a credit-card transaction to pay for the catalog item. The user's completed order is recorded 12 in the data base 11. Each step may be recorded in the database.

In the example depicted in FIG. 1b, the data base 11 is a simple legacy data structure.

Implementing the present invention in the exemplary embodiment described herein provides apparatus, systems and methods to convert the data in the data base 11 to an XML document which is then, in turn, transformed to a machine readable tree-based DOM.

FIG. 3 is a sequence diagram illustrating an exemplary business transaction between a business server and a partner server. Customer 1305 uses Web client 1055 to send response document request 1315 to business server 1015. The body of the response document request contains item selection 1310 and instructions for purchase of the selected item from an online catalog. The requested response document is an acknowledgment of the item selection and purchase instructions. The business server receives the response document request and parses 1320 out the item selection and purchase instructions from the response document request and creates a response. As part of the parsing step, the business server queries legacy database 1000 (FIG. 1) for item availability and pricing and composes a response document based on templates stored in Internet database 1010 (FIG. 1). The business server sends response document 1310 to the Web client and the Web client interprets 1335 the response document to create a display.

The business server notifies a business partner that a purchase order has been processed for the selected item. The business server sends email message 1355 to email server 1045 as part of a business partner notification process. Business partner 1300 uses email client 1060 to make email selection 1360 that is sent as email request 1370 to the email server. The email server sends the email message to the email client and the email client displays 1365 the email message to the business partner.

The business server orders new items from partner server 1030. To do so, the business server sends order document 1325 to the partner server. The partner server processes 1340 the order document and sends acknowledgment 1345 to the business server. The business server uses data contained in the acknowledgment to update 1350 the legacy database.

The business server translates data between legacy data systems and between disparate Internet data systems. FIG. 4 depicts data flow within two classes of software objects responsible for data translation operations within the business server. Adapter software object 1525 converts legacy data 1500 read from legacy database 1000 into XML document 1505 using the document description found in DTD 1510 for the XML document. The XML document is sent to other server objects 1515 for further processing. The adapter converts Internet documents into a format useful for updating the legacy database. The adapter object receives the XML document from another server object and converts the XML document into legacy data for storage in the legacy database. Alternatively, the adapter may create a DTD for the converted XML document when metadata is available for creation of the DTD. The adapter reads metadata 1520 and creates DTD 1510. The adapter uses the DTD to create the XML document from the legacy data read from the legacy database. Thus, the adapter uses the metadata to direct and control the translation of data between the legacy database and XML

document data structures.

Translator object 1700 translates one XML document into another XML document. The translator receives external XML document 1710 from external object 1705. The XML document may need to be translated into internal XML document 1715 before it is passed to other server objects 1515. The translation from an external XML format to an internal XML format allows a single set of internal server objects to process a wide array of external XML documents regardless of their format or source. The single set of internal server objects need only know how to manipulate a set of internal XML documents so long as each type of external XML document format is translatable into one of the known internal XML document formats.

Translation may be accomplished if a translation document in the form of an Extensible Style Language Transformation (XSLT) document exists for each translation. An XSLT document specifies how one XML document may be translated into another XML document. The translator reads in the external XML document and translates it using XSLT document document 1725 into the internal XML document. The internal XML document is then sent on to other server objects for use within the business server. The translator may also handle translations from internal XML documents into external XML documents in an analogous manner. The translator reads an internal XML document and uses a XSLT document document to translate the internal XML document into an external XML document.

FIG 5. is a depiction of a DOM object software object. The purpose of DOM object 1800 is to encapsulate a DOM representation of a XML document and expose a plurality of methods useful for manipulation of the DOM and consequently the XML document represented by the DOM. The DOM is contained within an internal data structure 1805. The internal data structure reflects the tree structure of the DOM and is in a suitable form where nodes and leaf elements may be added and deleted. The internal data structure is known as an infoset and embodies characteristics defined by the W3C Infoset working group. This information may be viewed at: http://www.w3.org/TR/xml-infoset. A plurality of methods for operation on the infoset are exposed for use by other software objects. Exemplary read 1810 method provides a way to populate the infoset by reading an XML document from a datastore. Exemplary add method 1815 provides a way to add new elements to the infoset. Exemplary delete method 1820 provides a way to delete elements from the infoset. Other software objects may invoke and modify a DOM object as a way of reading and modifying a XML document without knowing how each desired operation is actually performed on the XML document.

The adapter, translator, and DOM software object will be described in greater detail later within this specification, they are introduced here to facilitate discussion of the business server.

FIG. 6a is class diagram of a business server according to the present invention. The main control module in the business server is pipeline control module 1215, so named because document processing within the business server proceeds in a sequential manner through a collection of independent data flow paths termed pipelines. The pipeline control module is a

composite object composed of sorter class 1230, dispatcher class 1235, translator class 1240, and sender class 1245. The sorter class analyzes incoming documents to determine which pipeline should be used to process a particular document. The sorter class may use the translator class to create internal working documents, herein termed doclets, for use within the business server. The names and locations of the internal doclets are stored in dictionary 1220 used by all of the classes composing the pipeline control module. The dispatcher class uses doclets to create other doclets as the result of the application of business methods to the doclets. The business methods are encoded as business rules using a variety of programming languages such as Visual Basic (VB) and a variety of mark up languages such as XML. The dispatcher uses the translator class to create outgoing documents in a form suitable to the format of the incoming document. For example, if the incoming document was contained within a HTTP request from a client, the format for the appropriate outgoing document is a transmission document written in HTML that is transmitted back to the requesting client as a response. The outgoing documents are sent by sender class 1245 out to the Internet. The sender class contains methods suitable for a plurality of communications protocols. For example, the sender contains methods enabling the sender to format and send an outgoing document as an email message using SMTP as the communications protocol and the same sender contains methods to send an outgoing document as a HTTP response. Each of the classes composing the pipeline control module use methods in utilities class 1225 for various support functions as needed. The pipeline control class includes Web front end class 1200 to implement communications with other software objects over the Internet. Console front end class 1210 implements a user interface for control of the operations of the pipeline control module.

FIG. 6b is a data flow diagram illustrating intake of documents from the Web by the business server. Web server 1110 is used as a front end to business server 1105. The Web server accepts Internet documents written in several different formats. Preferably, each incoming document is composed at least partially in XML conforming to a DTD known to the business server such as RosettaNet document 1115 and Commerce One document 1120. Alternatively, an incoming document may be written using a custom DTD such as custom XML document 1125. The Web browser passes each incoming document to the business server. Alternatively, the Web server may pre-process the incoming documents by invoking local program 1100 to extract XML fragments from an incoming document and send the extracted XML fragments to the business server.

The path an incoming document takes through the business server is specified by documents containing instructions read by the objects comprising the business server. A process document specifies processing steps to apply to the incoming document and is herein termed a pipeline document. The instructions for matching a pipeline document with an incoming document are encoded in a server document.

FIG. 6c is a dataflow diagram of data through the business server. Business server 1400

instantiates and invokes a pipeline object 1405. The business server passes in an incoming document in the form of an XML document as well as request, response, and application objects 1425 from the business server environment. The pipeline creates dictionary object 1420 for the storage of doclets and DOM objects. A DOM object is instantiated and invoked. Once invoked, the DOM object encapsulates the incoming XML document. The DOM object is stored in the dictionary. Alternatively, sorter 1410 creates a DOM object to encapsulate the incoming XML document.

Control is passed by the pipeline to the sorter. The sorter examines the incoming XML document encapsulated in the DOM object and determines the correct path through the business server business processes for the incoming XML document by matching a pipeline document with the incoming XML document. The sorter uses the process document matching instructions to select the correct pipeline document based on the contents of the incoming XML document. The sorter makes a pipeline document selection and initializes the server environment as specified in the server document. For example, the sorter loads the pipeline document into the dictionary and translates the incoming document into a doclet that is stored in the dictionary.

Control passes back to the pipeline and dispatcher 1415 is invoked to processes the incoming XML document as defined in the pipeline document, placing any resultant processed documents into the dictionary.

FIG. 6d is a data flow diagram of the operations of a sorter object according to the present invention. Sorter 1610 accepts incoming XML document 1600 and creates a DOM object encapsulating the incoming document and stores the DOM object in dictionary 1615. The sorter uses a business server XML document 1620 to determine which of exemplary pipeline documents 1625 and 1630 should be used to specify the business process by which the incoming XML document is processed. The selected pipeline document is placed in server application collection 1635 for use by the previously described dispatcher.

Server documents and pipeline documents are composed in XML. In the case of a procedural language such as C, a function call with a void return value takes the form of 'foo(bar1,"bar");'. The '()', ',', '""', and ';' characters are syntactic elements that direct a compiler or interpreter to prepare a call to the function 'foo' passing in the variable 'bar1' and the value 'bar'. In a similar manner, tags in an XML document are used as syntactic elements to specify instructions for directing operations within a business server. For example the following XML fragment can be used to specify a call to the function 'foo': <Foo variable='bar1' > bar </Foo>. The attribute 'variable' is used to specify the input parameter passed as a variable and the text node 'bar' is used to specify the parameter passed as a character string.

Alternatively, the server and pipeline documents are preprocessed, as in an interpreted language such as VB Script, into bytecode, that is a highly optimized interpreted code.

Alternatively, the server and pipeline documents are compiled directly into machine code.

FIG. 6e is an exemplary business server XML document. The business server XML document describes the criteria used to select a XML pipeline document and translator XSLT document used by the sorter when setting up to process an incoming XML document. The exemplary business server XML document is composed of elements with each element defined by a start and end tag. The first tag 1900 identifies the document as a business server document. Business process tag 1910 signals the beginning of a definition of a business process element containing tests and processes used to process an incoming XML document. Name attribute 1915 names the process. Pipeline attribute 1905 identifies the pipeline XML document to be used to process the incoming document. Translator attribute 1920 identifies the XSLT document to be used to translate the incoming XML document. In the exemplary business process, the process is named "ABCPurchaseOrder" because it process a purchase order from ABC. The chosen pipeline is "plABCPurchaseOrder.xml" and the XSLT document is "ABCPurchaseOrder.xsl". Criteria tag 1930 begins the definition of the criteria element by an incoming XML document is tested to determine if the business process should be applied to the incoming XML document. Test batch tag 1950 begins the description of the first test batch element within the criteria to be applied the XML document. Test tag 1935 specifies the test. In this case, the XML document must contain an XPath in the XML document as specified in path attribute value 1925. The business process defined in business process tag 1960 illustrates an alternative process in which object tag 1955 begins the specification of an invocation of an object external to the sorter to determine if a business process is applied to the incoming XML document. The object specified by text node 1945 in the example is "OldStyleSorter.COMObjectName".

The XML server document comprises instructions used to Please consider that in the case of a function, say:

x=foo(bar1,bar2,"bar")

When a compiler program parses this, the '=', '(', ',' and "" characters are "tags" and "x", "bar1", "bar2" and "bar" are data. In this manner it is clear that the compiler interprets these as instructions. Our server andpipeline XML documents contain both keywords and separator characters(tags), as well as variables and constants (data), but these stillconstitute "instructions" to our server.If we describe it in this sense, then it becomes clear that these documentsare, indeed, instructions, and would in fact be preprocessed, much like aninterpreted language like VB Script or Java Script would be, into bytecode,or internal representation, which is a highly optimized interpreted code.This is what we would probably do (we've actually talked about doing thisfor performance reasons), although there is no reason why we couldn'tcompile directly

to machine code (except for the complexity and cost ofdoing so).

The following psuedocode specifies how the business server document of FIG. 6e is parsed by sorter 1610 (FIG 6b):

```
For each business process element in the server XML document:
    If there is an object element then:
        Invoke the object specified in the text node.
        Call the IsMyType() method of the specified object, passing in a
        dictionary and a DOM object containing an incoming XML document.
        The IsMyType() method returns TRUE if the incoming XML document
        is a proper document for processing by the processes of the current
        business element.
        If the return value is TRUE, then no more business elements are
        processed and the current business element passes.
    If there is no object tag, then:
        Find the criteria element.
        For each test batch element in the criteria element:
            For each test element in the test batch element:
                Use selectNodes() method with the XPath notation to
                retrieve a node-list from the DOM object encapsulating
                the incoming XML document.
                If the node-list is empty:
                    Skip all remaining test elements in the current
                    test batch element and go on to the next test
                    batch element.
        If all of test batch elements fail go on to the next business process.
If all business process elements fail:
    Throw an exception because the incoming XML document is not a valid XML
    document for processing by the business server.
If a business process element passes:
    Apply the XSLT document named in the business process tag translator attribute
    to convert the incoming XML document into an internal doclet.
    Store the internal doclet in the dictionary.
    If the pipeline XML document specified in the business process tag pipeline
    attribute is not cached in the application object:
        Read the pipeline XML object in and parse into a memory
        representation.
        Store the memory representation of the pipeline XML document in the
        application object.
```

Store the name of the pipeline XML document stored in the application object in the dictionary for programmatic reference.

Control is passed back to the pipeline object that instantiated and invoked the sorter object. The pipeline object then instantiates and invokes a dispatcher object. The dispatcher object modifies the doclets created from the incoming XML document by the sorter object using the processes defined in the pipeline XML document read and cached by the sorter object.

FIG. 6f is an exemplary pipeline XML document as used by a dispatcher. A pipeline XML document is composed of elements wherein each element is defined between start and end tags. Start tag 2100 indicates that this XML document is a pipeline XML document. Exception handler tag 2107 begins the definition of an exception handler element. Handler attribute 2105 names the exception handler element as "ADODB.ExceptionHandler". The exception handler element specifies the processes used to compose and send an exception message when the dispatcher encounters an error. Sender tag 2010 begins the definition of a sender element defining the type of sender used to send an exception message. Text node 2115 specifies that "ADODB.ExceptionSenderHTTPResponse" is the software object used to send an exception message as a response to a query. The name of the software object indicates that the exception message will be sent using HTTP as the communications protocol. An additional exception is sent via email as defined by text node 2120.

Component tag 2136 begins the definition of a component element. The component element specifies an operation performed using the incoming XML document. The pipeline XML document defines a series of component elements that tell the dispatcher which business operations to perform. The completion of all of the component elements within a pipeline XML document completes a transaction by the dispatcher. In the exemplary pipeline XML document, component tag 2136 begins the definition of a component element. Type attribute 2125 specifies the component element will be an internal database doclet. Target attribute 2130 names the internal database doclet as "MyDoclet". The dispatcher creates a doclet and stores the doclet in a dictionary as specified. The doclet is then available for use by other software objects through the dictionary. Text node 2135 specifies a XML document to be sent to a database server. The database server uses the XML document to populate the internal database doclet stored in the dictionary. The internal database document will contain data obtained by the database server from an Internet database populated with data suitable for direct conversion into Internet documents written in XML. Alternatively, a previously described adapter is invoked to populate the internal database doclet by drawing data from a legacy database. Alternatively, the internal database doclet is encapsulated within a DOM object as previously described.

Component tag 2140 begins the definition of a component element defining a business rule. A business rule is a series of operations applied to a data set to accomplish a particular

business transaction such as satisfaction of a purchase order. Type attribute 2146 specifies that the component element is a business rule. Text node 2145 specifies that the business rule is implemented by a software object invoked by the dispatcher. For example, a software object reads data out of an internal XML document that specifies a purchase order. The software object access an inventory database and determines if there are sufficient quantities of stock to satisfy the purchase order. If so, the software object issues a shipping order to a shipping department, issues a billing order to a billing department, and earmarks the stock in the database for use in satisfying the purchase order. In the exemplary pipeline XML document, the software object implementing a business rule is "ABCPO.MyCOMObject".

The internal database doclet is translated by the dispatcher as specified in a transform element. The dispatcher invokes a previously described translator and passes to the translator the name of the document to be translated and the XSLT document defining the translation. Component element tag 2160 begins the specification of a transform element. Type attribute 2150 specifies that the component element is a transform element. Source attribute 2155 specifies that the internal database document named "MyDoclet" is to be translated by the translator. Target attribute 2165 specifies that the result of the translation will be a translated XML document named "MyResponse". Text node 2170 specifies that the XSLT document used by the translator is "ABCPO_Rosettanet.xsl".

Component tag 2185 completely specifies a component element of type sender. A sender element specifies how the dispatcher invokes a sender object used by the dispatcher to send the results of a business transaction back to a client that posted the incoming XML document. Type attribute 2180 specifies that the component element is a sender element. Protocol attribute 2175 specifies the protocol to be used by the sender is for an HTTP response. Source attribute 2190 specifies the XML document sent by the sender is "MyResponse".

FIG. 7 is a cooperation diagram depicting the operations in business server during an exemplary business transaction. Web client 1700 posts an incoming document to Web server 1710. The document is written in a markup language and contains at least one XML fragment specifying a business transaction. The post from the Web client specifies a Web page to be read by the Web server from Web page storage 1715. The Web page contains instructions to remove the XML fragment from the incoming document to create an incoming XML document. The Web server sends the incoming XML document to business server 1720. The business server invokes pipeline 1725, passing in the incoming XML document. The pipeline creates incoming DOM object 1730 encapsulating the incoming XML document. The pipeline stores the incoming DOM object in dictionary 1760. The pipeline instantiates and invokes sorter 1745. The sorter reads in a business server XML document from server XML document storage 1735. The business server XML document defines the business operations to be implemented by the sorter as previously described. The sorter tests the incoming DOM object as previously described to determine which business processes need to be implemented. If a business process

is identified, the sorter reads in a pipeline XML document from pipeline XML document storage 1740 and stores the pipeline XML document in the dictionary. The sorter invokes incoming translator 1755 passing in a name for a XSLT document. The incoming translator reads the XSLT document from style sheet storage 1750 and uses the XSLT document to translate the incoming XML document encapsulated in the incoming DOM object into an internal doclet processed by other objects as specified by the pipeline XML document chosen by the sorter. The translator stores the internal doclet in the dictionary.

The pipeline invokes dispatcher 1770 to process the internal doclet created by the incoming translator using the business process specified by the pipeline XML document chosen by the sorter. The dispatcher reads the pipeline XML document and parses the pipeline XML document initiating the business processes specified within the pipeline XML document. The dispatcher invokes previously described adapter 1785. The adapter uses the services of database server 1790 to read data from a legacy database. The adapter uses the data to populate a working doclet stored in the dictionary. The dispatcher invokes rules object 1780 to perform the actual business transaction. The rules object creates working DOM object 1795 encapsulating the working doclet populated by the adapter. The rules object performs any necessary operations on the working DOM object to complete the business transaction. Modifications to the DOM object are immediately reflected in the working doclet stored in the dictionary. The adapter invokes outgoing translator 1775 passing in a name for an XSLT document specifying the conversion of the working doclet into an outgoing XML document. The outgoing document reads the outgoing XSLT document from style sheet storage 1750 and creates an outgoing XML document from the working doclet. The outgoing XML document is made available to other objects by placing the outgoing XML document in the dictionary. The dispatcher then invokes sender 1776. The sender routes the outgoing XML document as needed thus completing the business transaction.

Exposure of an XML Document as a Software Object

Manipulation of a XML document's equivalent DOM structure is facilitated in an object oriented environment if the DOM structure is an object with exposed interfaces instead of a static data structure. The present invention provides a facility for turning a DOM representation of a XML document into a software object, known as a DOM Object, consisting of two components: the Document component and the Node component. In an exemplary embodiment, a DOM object may be created from Microsoft XMLDOM XML documents. The XMLDOM control from Microsoft consists of several objects. The most commonly used are the IXMLDOMDocument and IXMLDOMNode objects. When an XML document is loaded, it is loaded into an XMLDOMDocument object. The DOM component wraps the XMLDOMDocument object with a document component object that exposes the properties and methods that apply to the XMLDOMDocument that it represents. (Note that reference herein

to "methods" that apply to XML documents means logic, program instructions and the like that define, interface with or provide an interface with various aspects of an XML document).

The present invention uses metadata to direct and control the translation of data between legacy and Internet data structures. In one embodiment, the present invention provides a method for converting legacy data to an Internet data structure. The Internet data structure may be characterizable by a tree-based structure definition. The legacy data to be converted may be characterized by a particular structure and by a set of relationships. The structure and the set of relationships that characterize the legacy data may be described by metadata comprising a plurality of metadata components.

The method to convert the legacy data to a Internet data structure comprises several steps. The adapter transforms the metadata into a set of DTDs and a corresponding tree-based structure definition. The metadata may contain the number of "columns" (also sometimes referred to herein as "record types") contained in the legacy data and may provide a description of the relationships between the columns and between the fields of each column.

The tree-based structure definition may comprise a plurality of node definitions. Each DTD declaration may be equivalent to one of the metadata components. Each of the node definitions may be equivalent to one of the metadata components.

The adapter identifies and stores each equivalent DTD declaration and the corresponding metadata component as conversion map components. The adapter identifies and stores each equivalent node definition and the corresponding metadata component as a conversion map component. The adapter then aggregates the conversion map components into a conversion map and uses the conversion map to map the legacy data into the Internet data structure.

Within this DOM component-wrapped object, the XMLDOM has, for each XML element or attribute, an XMLDOMNode object. When a new document is created from a template, the document that created is an "instance" of the template or class of document being created. To instantiate an object, an instance is made of the object that the template describes. Instantiating a DOM object automatically instantiates the underlying XMLDOM object. DOM and XMLDOM objects are classes with class definitions; the class definitions of DOM and XMLDOM objects function as templates. A class definition means a template that defines properties, methods, and in some cases, code procedures, that underlie an object. Accordingly, when a programmer wishes to use a DOM object, the programmer must use a class to instantiate the DOM object into a software object.

The DOM Object also has a Node component object that wraps the DOM component-wrapped object. Node component objects contain all the properties and methods needed to work with a specific node. The Node component object exposes a node in the underlying XMLDOM object. The DOM object treats elements and attributes in similar manner, to the extent possible. The creation of new elements and attributes are handled separately, however, to ensure that they

are created correctly. When a reference is made to a Node component object that is an attribute, the default property is the text property, which is the nodeValue of the underlying XMLDOM object.

The Document component object is what is first created when a new document is created. In an exemplary Microsoft XMLDOM embodiment of the invention, the Document component wraps the Microsoft XMLDOM XML tree by tag name, exposing Node component objects at each level. Most of the work happens at the Node component level. However, Node components are generally always attached to a Document component, directly or indirectly.

In an exemplary Microsoft XMLDOM embodiment of the invention described herein, mv.DOM and/or mvDOM represents the embodiment's DOM Object; mv.Document and/or mvDocument represents the embodiment's Document component object, and mv.NODE and/or mvNODE represents the embodiment's Node component object.

FIG. 8 is a graphic representation depicting the structural-element by structural-element wrapping feature of the invention in an exemplary Microsoft XMLDOM embodiment of the invention. As depicted in FIG. 8, a mvDocument 120 contains within it a XMLDOMDocument 121. The mvDocument 120 points to its highest level mvNode 122; the XMLDOMDocument 121 points to its highest level XMLDOMNode 123; the highest level mvNode 122 contains within it the highest level XMLDOMNode 123. The highest level mvNode 122 in the mvDocument 120 structure points to subordinate level mvNodes 124 and 126 respectively. The highest level XMLDOMNode 123 in the XMLDOMDocument 121 structure points to subordinate level XMLDOMNodes 125 and 127 respectively. The subordinate level XMLDOMNodes 125 and 127 are contained within the subordinate level mvNoes 124 and 126 respectively. As depicted in FIG. 8, the invention imposes on the mvDOM the structural features and characteristics of the underlying XMLDOM.

The syntax in an exemplary Microsoft XMLDOM embodiment of the invention for instantiating a Document component Object is as follows:

set obj=Server.CreateObject("GAX.mvDOM")

In the above CreateObject example, GAX.mvDOM is the ID of the class and obj is an instance of this class.

In an exemplary Microsoft XMLDOM embodiment of the invention, using the Document component object, the invention provides for the loading of an XML document from an existing Microsoft IXMLDOMDocument object, a URL, or an XML string; creating new root nodes; appending records with assigned attributes to efficiently link multiple related tasks; and validating the DOM object against its corresponding DTD or schema.

In an exemplary Microsoft XMLDOM embodiment of the invention, the Document component object has the following properties: 1.) A read-only property which exposes the Microsoft XMLDOM object so the properties that are exposed can be manipulated; 2.) A read-only property that exposes the Microsoft IDOM ParserError object; 3.) A read-only property

that exposes text representation of the XML document.

In an exemplary Microsoft XMLDOM embodiment of the invention, a number of methods are provided. Following are exemplary syntax and examples, using VBScript and/or JavaScript, for an exemplary embodiment of the invention providing interface and integration capabilities in an XML Internet environment/multivalue Legacy environment. In the embodiment described, mv.DOM and/or mvDOM represents the embodiment's DOM Object; mv.Document and/or mvDocument represents the embodiment's Document component object, and mv.NODE and/or mvNODE represents the embodiment's Node component object.

In the exemplary XMLDOM/multivalue embodiment, the following method syntax creates (the word create and the word instantiate are used synonymously) an mv.DOM object "GAX.mvDOM": 1.) In VBScript Syntax:

```
Object doc = Server.CreateObject("GAX.mvDOM")
```

and 2.) In JavaScript Syntax:

```
Var mvDocument = new ActiveXObject("GAX.mvDOM"):
```

In both examples above, the ActiveX control must have been previously registered.

Because Microsoft XMLDOM objects are an implementation (with some extensions) of the W3C DOM API Specification, one with ordinary skill in the art will understand that the invention provides the functionality described herein, and as depicted through the functionality provided by the embodiments described herein, with respect to any and all W3C DOM API compliant XML Parsers. The use of Microsoft XMLDOM is illustrative and not a limitation of the invention.

FIG. 9 provides a simplified, high level flow diagram depicting an exemplary embodiment of exemplary legacy data to Internet data conversion program logic flow. As shown in FIG. 9, the first step is to determine which table is the top-level table in the result-set 1001. The next step is to identify the number of columns the top-level table has 1002. The next step is to search the DTD to determine the number of, and to determine the identity of, elements that do not contain sub-elements or attributes and that can contain text that exist, or can exist, as a child element of the root element 1003. Next, match these lowest level elements, one for one with result-set columns from the top-level table 1004. The next step is to find the first child-table and count it's columns 1005. As depicted in FIG. 9, the next step is to find an element that can be repeated, which has sufficient elements or attributes to match the number of columns in this child table 1006. The next step is to match these repeatable elements, one for one, with the result-set columns from the child-table 1007. Next, repeat the process as described above for process blocks 1005 through 1007 to the Nth level 1008. Then, repeat the process as described above for process blocks 1002 through 1008 at every any level, for all child tables at that level 1009.

With regard to the logic flow described in FIG. 9, if an element contains attributes, and can contain text, load the element's attributes first, then the text node of the element itself. If

an element can be repeated, repeat it as often as allowed, then go on to the next element. E.g.: If there are three (3) columns, and the DTD defines an element that can occur 0-2 times followed by another element, create two of the first element, put the first two columns into them, and then create the next element, and place the third column's output into it.

It may be useful to view the above-described process, as depicted in FIG. 9, as a two part process. One part of the process is the identification of data elements at each level of the hierarchy of the data, and thereby, the identification of the overall hierarchical relationships of the data. This part of the process corresponds to the outer logic loop depicted in FIG. 9 as process blocks 1002 through 1009. The hierarchy of the data is used to define an axis, such as an axis of a matrix, along which the identified data elements can be partitioned as individual nodes for a tree data structure. The axis associated with this part of the process is, for instance a vertical, or y, axis.

In the case of relational database management legacy data, this data element identification part of the process consists of identifying child tables in the relational metadata on the one hand, and finding repeatable child elements in the DTD on the other hand. Child tables can exist within other child tables and can exist to any number of levels; multiple child tables can exist as siblings within the same parent table. Accordingly, in such a scenario, the present invention will follow multiple tree paths to build the XML data in a manner that properly preserves all of the relationships of the legacy data.

Another part of the process is the identification of data fields within a particular data element, or tree node. This part of the process corresponds to the inner logic loop depicted in FIG. 9 as process blocks 1005 through 1008. The data fields within each particular data element can be used to define a second axis, such as the horizontal, or x, axis of a matrix. This part of the process must be performed within each table and within each repeatable child element.

In the case of relational data base legacy data, this data field identification part of the process identifies the number of columns referenced from the relational data table and any tables joined with a standard inner or outer "join" to that relational data table. The data field identification part of the process then matches each of the columns identified to data elements that are at a data-containing level of the hierarchy and that correspond to an equivalent level of the table or tables being expressed.

FIG. 10 is a table representation of an exemplary set of business data. The data is represented in a hierarchical manner. The first column 701 represents information concerning Invoice Headers. The second column 702 represents information concerning Invoice Lines. The third column 703 represents information concerning Invoice Payments. The Invoice Number 704 is related to each of the columns 701, 702, and 703. There can be multiple Invoice Lines for each Invoice Header. There can also be multiple Invoice Payments for each Invoice Header. FIG. 11 is a graphical representation that depicts the relationships between each "column" of information.

FIG. 12 depicts an exemplary XML structure that represents the conversion of the data depicted in FIGS. 10 and 11 using the conversion method depicted in FIG. 9 and described above. The conversion did not result in any data loss.

The primary embodiment described herein for converting legacy data to an Internet data

structure uses as exemplary source legacy data, data stored using a simple data structure that provides for multivalued attributes. Examples of technologies that provide for the creation and maintenance of such multivalued data structures include among others PICK, ADP CORA, ADDS, Reality, and RealityX systems. Advanced Pick: Open Database and Operating System, Roger J. Bourdon, published 1996 and Pick for Professionals : Advanced Methods and Techniques, Harvey E. Rodstein, published 1990 are incorporated for all purposes herein by reference as if fully stated here and provide background information concerning multivalued data structures such as are provided in a PICK operating system.

The primary embodiment described herein for the legacy data to Internet conversion uses an exemplary XML document as the Internet data structure. Designing Distributed Applications With Xml : Asp Ie5 Ldap and Msmq, Stephen F. Mohr, published 1999, IE5 Dynamic HTML Programmer's Reference, Brian Francis, et al., published 1999, and XML IE5 Programmer's Reference, Alex Homer, published 1999 are incorporated for all purposes herein by reference as if fully stated here and provide background information concerning XML and DOM.

Conversion of Legacy Data into Internet Data

In FIG. 1, legacy database 1000 may be a simple legacy data structure. The present invention facilitates conversion of legacy data in the legacy database to an XML document which is then, in turn, transformed to a machine readable tree-based DOM.

FIG. 13 is a formatted representation of an invoice record in an exemplary simple legacy data structure. Certain display characters are used herein for purposes of discussion, including, for instance the "^", "]" and "\" characters. These characters are used to provide a visual format representations of the exemplary simple legacy data structure. The actual in-memory representation is, for instance, 254 on the ASCII chart for the character represented herein with the "^" display character; the actual in-memory representation is, for instance, 253 on the ASCII chart for the character represented herein with the "]" display character; and the actual in-memory representation is, for instance, 252 on the ASCII chart for the character represented herein with the "\" display character.

In the example depicted in FIG. 13, the Record ID 20 is identified by the root tag name of "Invoice" 21 and has a unique Invoice Number "00101" 22. The first 23 data field in the Record is the invoice date expressed as "11246" 24 which is a relative data format (the particular date format represented is the relative number of days since December 31, 1967). The second 25 data field is customer ID "ABC" 26. The third 27 data field is a multivalued data field containing two different product codes "XYZ" 28 and "XXX" 30, separated by a value delimiter "]" 29. The fourth 31 data field is a multivalued data field containing two quantity values "3" 32 and "1" 34 separated by a value delimiter "]" 33. The fifth 35 data field is a multivalued field containing two prices (each with two implied decimal points) "1495" 36 and "995" 38 separated by a value delimiter "]" 37.

The sixth 39 data field is a multivalued, multi-subvalued data field. The first value is comprised of two subvalues "Black" 40 and "220v" 42 separated by a subvalue delimiter "\" 41. The first value is separated from the second value by a value delimiter "]" 43. The second

value is comprised of three subvalues "Red" 44, "110v" 46 and "Battery" 48 each separated from the other by a subvalue delimiter "\" 45 and 47.

The seventh 49 data field is a multivalued data field containing two related Customer Check Nos."1023" 50 and "1076" 52 separated by a value delimiter 51.

The eighth 53 data field is a multivalued data field containing two check dates "11243" 54 and "11267" 56 separated by a value delimiter 55.

The ninth 57 data field is a multivalued data field containing two customer check amounts "1000" 58 and "1480" 60 separated by a value delimiter 59.

FIG. 14 is a representation of the way in which the record depicted in FIG. 12 is stored in memory. In FIG. 13, each data field is separated from the subsequent data field with a field delimiter "^" 100 - 108; no data field numbers 23, 25, 27, 31, 35, 39, 49, 53, and 57 as shown in FIG. 12 are present in the stored data. The ID 20 and Invoice 21 label tags are not actually part of the stored data but are depicted for reference of the reader.

The present invention provides apparatus, systems and methods for converting legacy data, such as the simple data structures exemplified in FIG. 14, to Internet data such as a tree-based rich data structure for which an exemplary illustration is depicted in FIGS. 15 and 16. FIGS. 15 and 16 represent for readability purposes only the converted record in an edited format.

To convert the data, adapter 1525 (FIG. 4) examines the legacy data for data that can be used in certain "generic" name tags, namely, "Records" as the root tag for the entire XML or mark up document; "Record" for the highest data hierarchy level; "Field" for the next subsequent data hierarchy level; "Value" for the next subsequent level data hierarchy level; and "SubValue" for the next subsequent data hierarchy level. The server is further programmed to use the notation "<tag name>" as the start-of-data tag notation; it uses the "</tag name>" as the end-of-data tag notation. For instance, "<Field>" designates the beginning of the data for a particular Field level; "</Field>" designates the end of the data for a particular Field level.

An empty tag can be annotated as follows: "<tag name/>". For instance, "<Field/>" denotes a tag that has no child elements. This notation can be used, even if the element contains attributes. For instance "<Field xml:space="preserve"/>" is a valid element that contains no child elements, even though it contains an attribute.

The adapter may receive as input alternative identifications with which the adapter overlays the generic name tags described above. This feature provides for a resulting marked up electronic document and the associated tree-based structure to be characterized by tag names that, as specified by a programmer, describe the data in terms that describe the business meaning of the data. For instance, the programmer can specify that the "Record" level be described as "Invoice." The generic tag name embodiment is described herein, but one of ordinary skill in the art of computer science will understand that the use of the generic tag name embodiment is not a limitation of the invention.

The adapter recognizes the delimiter "\" as marking a separation between SubValue data fields within the source legacy data record.

The adapter initializes the Internet data document with a root tag which in this case is "<Records>", to describe the data contained within the target object. The adapter then reads

a record from the legacy data, which for purposes of the example described here, is the record as depicted in FIG. 13. Then, for each record read, the server adds an empty tag, which in this example, is "<Record>" 201a. The adapter then adds an ID attribute to the "<Record>" 201a tag. The adapter sets the ID attribute 201b equal "=" to the data content 22 of the data record read that precedes the first field delimiter 100. This process is further explained below in more detail. As depicted in FIG. 15, the resulting tag contains: " Record ID="00101">" where "00101" is the data content 22 contained in the data record read as shown in FIG. 13 that precedes the first delimiter 100.

In order to assign a value to the ID attribute, the adapter looks for the first field delimiter in the form of "^" within the record. In the example record, the first field delimiter 100 is encountered by the adapter. In the embodiment described here, the adapter assumes that each legacy data record will contain a single Record ID and that the Record ID level will be the first field in the record. Accordingly, the adapter recognizes the delimiter "^" as marking the end of the Record ID level and as marking the end of each Field level within the source legacy data record. Accordingly, when the adapter encounters the first "^" delimiter in the record, the adapter inserts the start-of-data tag for the Record level "<Record>" 201a into the conversion record; the adapter uses the Record ID "00101" 22 from the legacy data record to build as an attribute of the "<Record>" 201a field, the ID attribute "ID="00101"" 201b and then inserts the ID attribute 201b into the "<Record>" start-of-data tag 201a.

Next, the adapter scans the source legacy record for the next "^" delimiter. After the first "^" delimiter 100 and prior to encountering the next "^" delimiter 101, the adapter recognizes as data the data field 24. Accordingly, the adapter inserts into the converted record the start-of-data tag notation for the data hierarchy level immediately subordinate to the "<Record>" level, which is the "<Field>" notation 202.

The adapter then encounters the next "^" delimiter 101 in the source legacy record. Accordingly, the adapter inserts the data 24 contained between the first "^" delimiter 100 and the next "^" delimiter 101 into the conversion record 203, and inserts immediately after the data 203 an end-of-data tag notation for the Field level of the hierarchy, "</Field>" 204.

The adapter continues to scan the source legacy record for the next "^" delimiter. After the "^" delimiter 101 and prior to encountering the next "^" delimiter 102, the adapter recognizes as data the data field 26. Accordingly, the adapter inserts into the converted record the start-of-data tag notation for the data hierarchy level immediately subordinate to the "<Record>" level, which is the "<Field>" notation 205.

The adapter then encounters the next "^" delimiter 102. Accordingly, the adapter inserts the data 26 contained between the "^" delimiter 101 and the next "^" delimiter 102 into the conversion record 206, and inserts immediately after the data 206 an end-of-data tag notation for the Field level of the hierarchy, "</Field>" 207.

The adapter continues to scan the source legacy record for the next "^" delimiter. After the "^" delimiter 102 and prior to encountering the next "^" delimiter 103, the adapter recognizes as data the data field 28. Accordingly, the adapter inserts into the converted record the start-of-data tag notation for the data hierarchy level immediately subordinate to the "<Record>" level, which is the "<Field>" notation 208.

The adapter then encounters a "]" delimiter 29. The adapter is programmed to recognize the delimiter "]" as marking a separation between Value data fields within the source legacy data record. Accordingly, the adapter inserts into the converted record the start-of-data tag notation for the data hierarchy level immediately subordinate to the "<Field>" level, which is the "<Value>" notation 209.

Having encountered the "]" delimiter 29, the adapter inserts the data 28 contained between the "^" delimiter 102 and the "]" delimiter 29 into the conversion record 210; inserts immediately after the data 210 an end-of-data tag notation for the Value level of the hierarchy, "</Value>" 211; and inserts into the converted record a second start-of-data tag notation for the data hierarchy level immediately subordinate to the "<Field>" level, which is the "<Value>" notation 212.

Next, the adapter encounters the "^" delimiter 103. Accordingly, the adapter inserts the data 30 contained between the "]" delimiter 29 and the "^" delimiter 103 into the conversion record 213; inserts immediately after the data 213 an end-of-data tag notation for the Value level of the hierarchy, "</Value>" 214; and inserts into the converted record the end-of-data tag notation for the data hierarchy level immediately subordinate to the "<Record>" level, which is the "</Field>" notation 215. The process described above is referred to herein as the multivalue conversion.

As can be seen from the description of the Record, Field and Value level conversions, the invention provides for the creation of successively nested levels in the converted record corresponding to each subordinate level of the legacy data structure hierarchy.

For explanatory purposes, the embodiment described herein uses a serial processing approach. Alternative embodiments of the invention make multiple scans of each record to establish, first, levels of data within the converted record for all of the highest levels of the legacy hierarchy; second, levels of data within the converted record for the second level of the legacy hierarchy; and so on. Such alternative embodiments insert data into the converted record using the same nesting structure as described above.

In a manner similar to the multivalue conversion described above, the adapter converts the legacy data, beginning with the "^" delimiter 103, including the multivalued fields 32 and 34 separated by the multivalue delimiter 33 to the converted data fields 216 through 223. And similarly, the adapter converts the legacy data, beginning with the "^" delimiter 104, including the multivalued fields 36 and 38 separated by the multivalue delimiter 37 to the converted data fields 224 through 231.

The invention is extensible to any number of levels of hierarchy. In the example depicted in FIG. 13, and in the embodiment described herein, the adapter converts subvalue levels of data as are represented in the source legacy data beginning with the "^" delimiter 105. multivalues are separated with the delimiter "]", such as the "]" delimiter 43; whereas multi subvalues are separated by the delimiter "\", such as with the "\" delimiters 41, 45, and 47. In subvalue conversion, the adapter recognizes the "\" delimiter as separating subvalues within a value field. Accordingly, the adapter inserts into the converted record the start-of-data and end-of-data Value tag notations, e.g., 233 and 240 respectively, and nest within the start-of-data and end-of-data Value tag notations, e.g., 233 and 240 respectively, the appropriate subvalue data,

e.g., 235 and 238 with the appropriate start-of-data and end-of-data SubValue tag notations, e.g., 234 and 236, and 237 and 239 respectively.

In a manner similar to the multivalue conversion described above, the adapter converts the legacy data, beginning with the "^" delimiter 106, including the multivalued fields 50 and 52 separated by the multivalue delimiter 51 to the converted data fields 253 through 260; the adapter converts the legacy data, beginning with the "^" delimiter 107, including the multivalued fields 54 and 56 separated by the multivalue delimiter 55 to the converted data fields 261 through 268; the adapter converts the legacy data, beginning with the "^" delimiter 108, including the multivalued fields 58 and 60 separated by the multivalue delimiter 59 to the converted data fields 269 through 276. When the adapter encounters an end of record in the source legacy data, the adapter inserts into the converted record an end-of-data tag notation for the Record level, "</Record>" 277.

The Internet representation of the data, for instance as depicted in FIGS. 15 and 16, uses both beginning and ending delimiters in the form of tags to designate the beginning and ending of a data element. On the other hand, multivalued data, as depicted in FIG. 14, uses ending delimiters to designate the end of a data element. The multivalued data delimiter positions are implicit metadata. Accordingly, each entire multivalued data record might need to be scanned in order to determine the structure of the record and of the data within the record.

The result of the legacy data to Internet data conversion as described thus far, in the embodiment and the example described herein, is an XML document. References herein to XML documents are understood to be electronic documents that contain an XML mark up schema. The XML document is not, however, in a tree-based form. Rather, an XML document must be translated to some tree-based form, such as a DOM.

A DOM does not contain any of the start-of-data or end-of-data tag notations described above for the XML document. Rather, a DOM is a tree-based rich data structure.

In an embodiment, such as the one described herein, where the result of the legacy data to Internet data conversion is an XML document, a standard XML parser can be used to parse the data from the XML document to a tree-based representation in memory.

One feature of the above-described aspect of the invention is that the adapter does not need to know any particular information, or any naming conventions, about the legacy data in order to convert the legacy data to an Internet data structure. Rather, the conversion was done using generic naming conventions on the Internet side, and without any knowledge of the naming conventions on the legacy side. Nevertheless, the converted XML document preserves all of the data and all of the relational and hierarchical relationship intelligence within the legacy data. As will be discussed further below, other aspects of the invention methods to label the resultant Internet data structure with descriptive tag names.

It should be understood that the example simple data structure multivalued legacy data is used herein for illustrative purposes only and is not a limitation of the invention.

multivalue systems have an implicit metadata in that the metadata is discovered at the time that the data is read. Because the metadata is implicit in multivalue records, the metadata can change, to some extent, on a record-by-record basis, because the metadata is expressed in each record's structure. Furthermore, the metadata in a multivalue system may provide certain

pre-defined limitations. For instance, some multivalue structures may only provide three (3) levels of nesting (e.g., attribute, value, subvalue); some multivalue structures may provide for an arbitrary number of fields.

In integrating legacy systems and data with Internet systems and data, many of the features of the present invention serve functions in both directions of such integration efforts. The following syntax and examples of detailed features of the invention are provided for illustration of the integration features. However, one with ordinary skill in the art of computer science will understand that inclusion of the description of these features at this particular point in the description of the invention is for illustrative purposes and does not limit the use of such features to a particular direction in such integration efforts.

In an exemplary XMLDOM/multivalue embodiment, the following Load method syntax uses mvDocument to load an existing XML document from an existing IXMLDOM object, a URL, or an existing XML string:

LoadDOM (IXMLDOM).

Executing a method using the above exemplary LoadDOM method syntax creates an mv.DOM object from an existing Microsoft IXMLDOMDocument object; it does so in the exemplary embodiment without re-parsing. An exemplary Boolean string is returned by the method to show whether the document loaded successfully or not. The method returns a value of true if the document loads successfully; false otherwise. The example as depicted below and as depicted in FIG. 17 is illustrative:

```
set domObject=Server.CreateObject("Microsoft.XMLDOM")
domObject.load("http://myurl.com/foo.xml")
set obj=Server.CreateObject("GAX.mvDOM")
If Not obj.LoadDOM(domObject) Then
   msgBox "Problem loading document!"
End If
obj.foo.Data="Hello"
Response.Write("<p>"&domObject.selectSingleNode("Data").
        value & " World!")
```

As depicted in FIG. 17 and in the above exemplary illustration, operations on and/or changes to an mvDOM object actually effect the values contained in a Microsoft XMLDOM object underlying the mvDOM object. As depicted in FIG. 17, an empty instance, domObject 130a, of a MicroSoft XMLDOM object, "Microsoft.XMLDOM" 131, is created 130. The Microsoft XMLDOM object is then loaded 132 with an XML Document located at URL "http://myurl.com/foo.xml" 133. An empty mvDOM object 134a, named "GAX.mvDOM" 135, is then created. In creating the empty mvDOM object 134a, named "GAX.mvDOM 135,

the default structure of GAX.mvDOM is established as an instance of an IXMLDOM object. Next, the mvDOM object 136a (same as 134a) is loaded 137 using the Microsoft XMLDOM object, domObject 138 (same as 130a) structure as the internal structure of the mvDOM object. The LoadDOM method replaces the mvDOM object structure with the Microsoft XMLDOM object structure. The LoadDOM method returns a Boolean string that is tested 136 to determine whether the document loaded successfully or not. As depicted in FIG. 17, if the method returns a value of valse, then the document was not loaded successfully, and an error "139b" is reported in the message box 139a. A character string, with the value of "Hello" 142 is assigned to the obj.foo.Data node of the mvDOM. Then, a message is written using as the message source the Microsoft XMLDOM object 143a (same as 130a) node "Data" 143b concatenated with the word " World!". If executed, the sample code would write the words "Hello World!" — showing that the changed value assigned to the mvNode Data in the mvDOM object actually effected the value of the XMLDOMNode Data value contained in the Microsoft XMLDOM object.

Changes to the mvDOM object 'obj' instantly affect the contents of the DOM object 'domObject'. This is because the loadDOM method wraps the passed-in XMLDOM object by reference.

In the exemplary XMLDOM/multivalue embodiment, the following Load (URL String) method syntax loads the XML document specified by the URL synchronously, and validates it against the XML-schema or DTD referenced within the document (if any):

Load (URL String)

The following example is illustrative:

```
boolValue = doc.load("e:\myinvoices\amazon.xml")
```

In the above example, a URL string, "e:\myinvoices\amazon.xml", contains the URL to an XML document or an Active Server Page or Servlet that generates an XML document. References to URL's to, among others, an Active Server Page, or to a Servlet that generates an XML document, are illustrative of the types of technologies to which URL strings, as used in the examples described herein, refer, and are not a restriction of the invention. A Boolean string is returned to show if the document loaded successfully or not. The following example is further illustrative:

```
if not obj.load("http://myserver.com/
myData.xml") then
    ' handle error
end if
```

In the exemplary XMLDOM/multivalue embodiment, the following LoadXML (XML String) method syntax loads an XML document from a string:

LoadXML (XML String)

The XML string is defined as a representation, in ASCII text, of an XML document. The document is loaded synchronously, and is validated against the XML schema or DTD referenced within it (if any). If there is no referenced schema, the document must only be well-formed. A Boolean string is returned to show if it succeeded in loading or not. The following example is illustrative:

```
If Not obj.loadXML("<MVData><Record Id=""foo""><Field>bar
</Field></Record></MVData>") Then
    msgBox "Problem loading document!"
End If
```

The following example is further illustrative:

```
boolValue = doc.loadXML("<Invoice><InvoiceNumber>123
    </InvoiceNumber></Invoice>)
```

In the immediately preceding example, the document to be loaded and validated from an XML string is InvoiceNumber 123.

The invention provides a Post method, an exemplary syntax of which is as follows.

```
set newmvDomObj = obj.post(
"http://www.myserver.com/writeorder.asp", ["userid"][,      "password"])
```

Executing a Post method as depicted above sends the XML object represented by the current mvDocument object to a specified URL for processing. Executing the above Post method returns a new mvDomDocument containing the response XML from the URL that was accessed.

In the exemplary XMLDOM/multivalue embodiment, the following createRoot method syntax creates a new Root Node document element where RootName is the name of the root element of the new document:

```
obj.createRoot("mvData")
```

Executing the above method returns the mvNode of the new object or document. Returning as a return value the object upon which the activity operated, in this case, the mvNode named "mvData" makes the mvNode "mvData" available to the next activity. This feature of the invention provides the further functionality of being able to link multiple activities together in a single line of programming code. A "dot" notation syntax is used in an exemplary embodiment to link multiple activities together in a single line of code. The following example is illustrative:

```
obj.createRoot ("mvData").append ("Record").
        appendAttribute  ("Id)  = 1
```

In the above example of the createRoot method, a new root node <MVData>, is created, a <Record> child node is appended, an attribute called "Id" is added to the Record node, and a value of "1" is assigned to the Id attribute.

The XML data for the above example is as follows:

```
<MVdata>
    <Record Id="1"/>
</MVData>
```

As compared to the single line of code using the above-depicted exemplary createRoot method syntax as provided by the invention, the equivalent code with Microsoft's XMLDOM requires the several lines of code depicted below:

```
obj.loadXML("<MVData >")
set attrNode=obj.createAttribute("Id")
attrNode.nodeValue="1"
set eltNode=obj.createElement("Record")
eltNode.setAttributeNode(attrNode)
obj.selectSingleNode("MVData").appendChild(eltNode)
```

In the exemplary XMLDOM/multivalue embodiment, the following method series syntax example, which is also depicted in FIG. 18, validates the current mv.DOM object against its DTD or schema. A value of true is returned if the object is validated; false if it fails validation. The exposed domParseError property is interrogated to determine the reason for validation failure.

```
s = "<!DOCTYPE MVData [“
s = s & “<!ELEMENT MVData (Record*)>”
s = s & “<!ELEMENT Record (#PCDATA)>”
s = s & “]>”
s = s & “<MVData><Record/></MVData>"
obj.loadXML(s)
obj.MVData.append("foo")
if obj.validate() then
   msgBox “Data is valid”
```

```
else
   msgBox "Data is NOT valid"
end if
```

For the above validation syntax example, and as depicted in FIG. 18, the DTD indicates that an MVData object must contain a Record element only. A record element "s" is built 150a-150e and loaded 151a into the MVData object 151b with a Record element 151a-1. A 'foo' element 152a is appended 152bd to the Record 152c of the object 152d. A "foo" element, however, is not valid for this DTD. Therefore, the validation method 153a response returned is "false", resulting in a message that the "Data is NOT valid" 156.

In the exemplary XMLDOM/multivalue embodiment, the mvDocument object has a number of properties which can be exposed using methods as described below. For example, the following domDocument method syntax example exposes the underlying Microsoft XMLDOM object, allowing use of any functionality of Microsoft XMLDOM object:

```
set myDom=obj.domDocument
```

The domDocument property is read-only. However, any read/write properties that the domDocument property exposes can be set.

As another example of mvDocument properties, the following domParseError method syntax example exposes the Microsoft IDOMParseError object:

```
set myParseError=obj.domParseError
```

The domParseError property is read-only. The parameters of the domParseError method are explained below:

errorCode	Returns the error number or error code as a decimal integer.
fileposline	Returns the number of the line in the document that contains the error.
linepo	Returns the character position of the error within the line in which it occurred.
reason	Returns a text description of the source and reason for the error, and can also include the URL of the DTD or schema and the node within it that corresponds to the error.
srcText	Returns the full text of the line that contains the error or an empty string if the error cannot be assigned to a specific line.
url	Returns the URL of the most recent XML

document that contained an error.

The domparseError property can also be exposed from the underlying DOM object using the syntax example below:

```
set myParseError=obj.domDocument.parseError
```

As another example of mvDocument properties, the following "xml" method syntax example exposes a read-only text representation of the XML document:

```
String mvDocument=obj.xml
```

Continuing with the exemplary Microsoft XMLDOM embodiment of the invention which provides interface and integration capabilities in an XML Internet environment/multivalue Legacy environment, a number of mvNode methods are provided. One such method is the appendAttribute method. The appendAttribute method creates a new attribute in an existing element and returns the mvNode to that element. An exemplary syntax is as follows:

```
obj.MVData.appendAttribute("xml:space")="preserve"
```

As depicted in the examples below and in the before XML fragment illustrated in FIG. 19 and the after XML fragment illustrated in FIG. 20, the appendAttribute example method depicted above creates a new attribute, "xml:space" 160b, in the MVData element 160a and assigns 160c it a value of "preserve" 160d. To illustrate the effect of this method, the following XML fragment is illustrative of an existing element (the "before" XML fragment, as is also depicted in FIG. 19) prior to execution of the method:

```
<MVData>
<Record>
    <Field>
        I'm first!
    </Field>
    <Field>
        new data
    </Field>
</Record>
</MVData>
```

Once the above example appendAttribute method is executed, the same XML fragment appears as follows with the attribute of the MVData element 160a "xml:space" 160b, and in FIG. 20, set equal to 160c the value "preserve" 160d:

```
<MVData xml:space="preserve">
<Record>
 <Field>
        I'm first!
    </Field>
    <Field>
        new data
    </Field>
</Record>
</MVData>
```

Another mvNode method in the exemplary Microsoft XMLDOM embodiment is the append Element method. The append Element method creates a new element (a Child element) under an existing mvNode with the name provided in the elementName parameter and returns an mvNode. The following syntax example is illustrative:

```
obj.MVData.Record.append("Field")="new data"
```

The append Element syntax example above, as depicted in FIG. 22, creates a new Field element 162,164 under the existing Record element 161, 165, in the existing MVData element 160a, 166, and inserts the new Field at the first level after the Record element 161; then assigns the value "new data" 163 to the mvNode that the append method returned.

Note that the exemplary embodiment described here behaves in a manner consistent with the behavior of Visual Basic. Accordingly, assigning a value to an mvNode object with no specification of a property to be assigned to the string, results in a default selection of the "text" property being assigned to the value. This is normal behavior for Visual Basic objects. Someone with ordinary skill in the art will understand that this behavior is specific to Visual Basic and not a limitation of the invention.

To illustrate the effect of the append Element method, the following XML fragment is illustrative of the existing element prior to execution of the method:

```
<MVData>
<Record/>
```

```
</MVData>
```

Once the above example append (elementname) method is executed, the resulting XML representation is, as is also depicted in FIG. 21, the following:

```
<MVData>
<Record>
    <Field>
        new data
    </Field>
</Record>
</MVData>
```

Some mvNode methods expose certain mvNode properties, including, for example: count, domNode, name, and text.

The count method returns the count of the number of junior sibling nodes with the same tag name, including the referenced sibling. An example mvDom object 'obj' containing the following XML object is illustrative of a subject of the count method:

```
<MVData>
    <Record Id="1"><Field>one</Field></Record>
    <Record Id="2"><Field>two</Field></Record>
    <Record Id="3"><Field>three</Field></Record>
</MVData>
```

The example count methods below, each operating on the above example XML object, return the values indicated below:

obj.MVData.Field.count	(Returns the value of 3).
obj.MVData.Field(1).count	(Returns the value of 3).
obj.MVData.Field(2).count	(Returns the value of 2).
obj.MVData.Field(3).count	(Returns the value of 1).

Another mvNode method in the exemplary Microsoft XMLDOM embodiment that exposes an mvNode property is the domNode method. The domNode method exposes an interface with the IXMLDOMNODE of an existing node in the underlying XMLDOM

object. Once this IXMLDOMNODE interface is exposed, the underlying DOM object can be accessed. The IXMLDOMNODE interface is a read-only property, but exposes some read/write properties.

Another mvNode method in the exemplary Microsoft XMLDOM embodiment that exposes an mvNode property is the name method, which exposes the name of an existing node as a string. The name property is read-only. Accordingly, the name method cannot be used to rename an element or attribute.

Another mvNode method in the exemplary Microsoft XMLDOM embodiment that exposes an mvNode property is the text method, which exposes the text value at an existing node allowing the text value of the node to be reset. In the exemplary microsoft XMLDOM embodiment, the text method assumes that there is only one text value in a single node.

Using mvNode methods and syntax as was previously explained above with regard to the exemplary Microsoft XMLDOM/multivalue environment embodiment, the following can be determined: the total number of child elements with the same parent and the same tag name; and the total number of child elements, regardless of tag name.

In the exemplary Microsoft XMLDOM/multivalue environment embodiment, a childNodes method is provided for getting a collection of child elements with matching tags. The childNodes method returns a collection of all child element nodes of the specified parent element that have tag names that match the value of the name parameter. The parent element is defined by the current mvNode object. If the current mvNode object is an attribute, an error is generated. The tag name of each child object must match the value of the name parameter in order to be included in the collection returned by the childNodes method and may include a namespace qualifier. An exemplary syntax of the childNodes method in the exemplary embodiment is:

```
mvNode.obj = childNodes ([name parameter])
```

The name parameter consists of a String. Provided the name parameter String value is present, childNodes method returns either a list of nodes with the tag name indicated in the name parameter String or a NULL value if the call fails. The following example is illustrative:

```
For Each MVRec In obj.MVData.childNodes("Record")
Response.Write("<p>Next Id=" & MVRec.Id.text)
Next
```

The childNodes example above will print a list of Id's of "Record" elements in the

MVData root element of the obj mvDom object.

If a name parameter String is not specified, the childNodes method returns a collection of all child element nodes of the specified parent element. If the parent element is an attribute, an error is reported. If the method fails to find any child elements for the specified parent, a NULL value is returned. The following example is illustrative:

```
ObjectLines = doc.invoice.childNodes()
```

In this above example, no name parameter String value is present. Accordingly, the childNodes method returns a list of all child nodes of the parent element invoice.

In the exemplary Microsoft XMLDOM/multivalue environment embodiment, an insertBefore method is provided for inserting a new sibling element before the current element, with the name specified in the elementName parameter. The insertBefore method returns an mvNode. The following example is illustrative:

```
obj.MVData.Record.Field(1).insertBefore("Field")

="I'm first!"
```

The insertBefore example above creates a new Field element, and inserts the new Field element before the 1st Field element; the method then assigns the value "I'm first!" to the new mvNode that the insertBefore method returns. The following XML fragment is a snapshot prior to executing the above exemplary insertBefore method:

```
<MVData xml:space:"preserve">
    <Record>
     <Field>
    new data
    </Field>
    </Record>
</MVData>
```

Once the above InsertBefore (elementname) example method is executed, as depicted in FIG. 24 new Field element 167 and 169 is added to the Record 161 and 165, with the value of "I'm first!" 168, at an mvNode level subordinate to the Record element 161 and 165, at the same "sibling" level as the prior-existing Field element 162 and 164, but occurring before the prior-existing Field element 162 and 164. The resulting XML representation is as follows and as is also depicted in FIG. 20:

```
<MVData xml:space="preserve">
    <Record>
    <Field>
        I'm first
    </Field>
    <Field>
        new data
    </Field>
    </Record>
</MVData>
```

In the exemplary Microsoft XMLDOM/multivalue environment embodiment described here, a remove method is provided for removing the current node from the document. The remove method does not provide any return value. The following example is illustrative:

```
obj.MVData.Record.Field(1).remove()
```

This example remove method above removes the first Field element and any and all nested elements and attributes from the Record element. The following XML fragment, and as depicted in FIG. 20, is illustrative of an XML object prior to execution of the above remove method example:

```
<MVData xml:space="preserve">
<Record>
<Field>
    I'm first!
</Field>
<Field>
    new data
</Field>
```

```
</Record>
</MVData>
```

Once the above remove method example is executed, the same fragment appears as is depicted in FIG. 22 and as follows:

```
<MVData xml:space="preserve">
    <Record>
    <Field>
        new data
    </Field>
    </Record>
</MVData>
```

In the exemplary Microsoft XMLDOM/multivalue environment embodiment, an exists method is provided for searching the current element for any immediate child element that has the name specified in the search criteria. The exists method returns a value of true if a match is found; otherwise it returns a value of false. The following example is illustrative:

```
if not obj.MVData.exists("Record") then
msgBox "No items in your data!"
end if
```

The above exists method example checks to see if there are any Record elements immediately under the MVData element. The XML fragment as depicted in FIG. 20, and as depicted below, is illustrative of an XML object prior to the execution of the above example exists method:

```
<MVData xml:space="preserve">
    <Record>
    <Field>
```

```
                I'm first!
            </Field>
            <Field>
                new data
            </Field>
            </Record>
        </MVData>
```

In the above XML fragment, an element called Record exists. Accordingly, the if statement is not true and the message box message "No items in your data!" will not be generated.

Internet Data to Legacy Data Conversion and Integration

FIG. 23 is a graphical representation of an exemplary tree-based rich data structure in an exemplary Internet data structure, which, in the case depicted, is a DOM of the data depicted in FIGS. 15 and 16.

The present invention provides a method for converting Internet data to a simple legacy data structure. The Internet data structure may be characterized by a tree-based structure definition. The legacy data structure may be characterized by a particular structure and by a set of relationships. The structure and the set of relationships that characterize the legacy data may be described by metadata comprising a plurality of metadata components.

The method to convert the Internet data to the legacy data structure comprises several steps. Adapter 1415 (FIG. 7) transforms the metadata into a set of DTD declarations and a corresponding receiving tree-based structure definition. The receiving tree-based structure definition comprises a plurality of receiving node definitions. Each DTD declaration is equivalent to one of the metadata components. Each of the receiving node definitions is equivalent to one of the metadata components.

The adapter stores each equivalent DTD declaration and the corresponding metadata component as conversion map components. The adapter stores each equivalent receiving node definition and the corresponding metadata component as a conversion map component. The adapter then aggregates the conversion map components into a conversion map. The

adapter uses the conversion map to create a receiving tree-based structure.

The adapter then compares the Internet data tree-based structure with the receiving tree-based structure. For each node in the Internet data tree-based structure, the adapter identifies the equivalent receiving tree-based structure node. The adapter builds a translation table of the equivalent Internet data tree nodes and the corresponding receiving tree nodes. The adapter uses the translation table to map the Internet data to the receiving tree-based structure. Then, using the conversion map, the adapter translates the data in the receiving tree-based structure into the legacy data structure.

To illustrate the invention, a multivalue data structure is used as an exemplary legacy data structure. The way in which the exemplary embodiment of the invention converts from the tree-based DOM structure used herein to illustrate the invention, to the exemplary multivalued simple legacy data structure is to traverse the tree, beginning with the highest level node 600.

In describing the Internet data to legacy data conversion, references to the source Internet data are made with reference to FIG. 23b, references to the resulting legacy data are made with reference to FIG. 13. It will be understood by one reasonably skilled in the art that the conversion from the Internet data structure to the legacy data structure is basically the reverse process of the legacy data to Internet data conversion. Therefore, only the first steps of the exemplary embodiment of the Internet data to legacy data conversion is described in detail below.

The adapter asks the highest level node 600 to list all of its immediate children. In response, the adapter obtains a list of all of the children to the highest level node, e.g., 602 and 603 are depicted as options, because the details for those records will not be discussed herein, and 601. The adapter will traverse the tree one Record node at a time. Beginning in our example with Record node 601, the adapter initializes a legacy data record for the data contained in the Record node 601 branch; the adapter inserts the ID, "00101" attribute of the Record node 601 into the beginning of the new legacy data record 22, and inserts a "^" delimiter after the attribute 100.

Further traversing the tree, the adapter asks the Record level 601 of the tree to identify all of its children. In response, the adapter obtains a list of all of the children of the Record level 601 of the tree. In this case, the children of the Record level 601 are listed as, among others, Field nodes 604, 606, 608, 613, and 626. The adapter will traverse the tree one Field node at a time.

Beginning with the first Field node 604 subordinate to the Record node 601, the adapter finds that only one data field "11246" 605 is provided by Field node 604.

Accordingly, the

adapter inserts the data field "11246" 605 into the new legacy record 24.

The adapter then encounters the next Field node 606 and immediately inserts a Field delimeter "^" 101. The adapter recognizes the single data element "ABC" 607. Accordingly, the adapter inserts the data field "ABC" 607 into the new legacy record 26.

The adapter provides a number of conversion and integration capabilities, including, among others, methods that delete records from a file (e.g., a method named "DeleteXML" in an XML Internet environment), read records from a file (e.g., a method named "ReadXML"in an XML Internet environment), and write records to a file (e.g., a method named "WriteXML"in an XML Internet environment). An exemplary embodiment of these adapter methods is provided below using, as an illustrative example, adapter methods that integrate data and functionality between an XML-based Internet environment and a multivalue Legacy environment, using Visual Basic script.

A DeleteXML method provides the capability to delete a record from an XML file that is to be written to a Legacy file, which in the exemplary embodiment depicted here, is a multivalue file. The DeleteXML method uses the first Field element in each Record element as the itemId of the item to delete from the file named as the method property. An exemplary method syntax for the DeleteXML method is:

```
obj.DeleteXML fileName, xmlString
```

The parameters noted in the above syntax are defined as follows:

fileName	The name of the multivalue file to which to write the data.
xmlString	The String containing the XML document. The syntax only requires that the record element with an id attribute be specified.

An example of the DeleteXML method is as follows: obj.DeleteXML "INVOICES","<MVData><Record Id = ""00123""/></ MVData>"

The above example DeleteXML method deletes the Record Id 00123 from a file named the INVOICES file. According to the example, and consistent with the multivalue DTD and content model, if the record does not exist, no error message is generated.

A ReadXML method returns an XML object from a data source, which in exemplary embodiment depicted here, is a multivalue data source. The file name is in the method property. The result is returned as XML output. The Syntax for the exemplary method is:

xmlString=obj.ReadXML(fileName[,filterText[,maxItems]])

The parameters noted in the above syntax are defined as follows:

fileName	The name of the multivalue file to get the data from.
filterText	An optional TCL method that activates a SELECT-LIST. If it is omitted, all items are selected.
maxItems	The maximum number of items to retrieve.

An example of the ReadXML method is as follows:

```
string XML = obj.ReadXML("INVOICES","SELECT INVOICES WITH
CUSTCODE = ""ABC""",10)
```

In the exemplary Microsoft XMLDOM/multivalue environment embodiment depicted here, the above ReadXML example reads a multivalue record such as the one depicted below:

```
>ED INVOICES 00123
00123
TOP
.L22
001 11256
002 ABC
003 X]N
004 1]2
005 BLACK\220V]RED
EOI 005
```

In the exemplary Microsoft XMLDOM/multivalue environment embodiment depicted here, the above ReadXML example returns a BSTR variable which contains the XML representation as shown below:

```
<MVData>
  <Record Id="00123">
    <Field>11256</Field>
    <Field>ABC</Field>
    <Field>
```

```
            <Value>X</Value>
            <Value>N</Value>
        </Field>
        <Field>
            <Value>1</Value>
            <Value>2</Value>
        </Field>
        <Field>
            <Value>
                <Subvalue>BLACK</Subvalue>

                <Subvalue>220v</Subvalue>
            </Value>
            <Value>RED</Value>
        </Field>
        </Record>
</MVData>
```

In the exemplary embodiment depicted here, the above ReadXML example reads its data from the INVOICES file after invoking a SELECT-LIST with the multivalue SELECT method shown. This method gets all INVOICES items which CUSTCODE equal to ABC. Lastly, the example returns only the first 10 items that it encounters.

A WriteXML method writes a raw XML message to a data source, which in the exemplary embodiment depicted here, is a multivalue data source. The XML input contains the XML string to write and the filename is in the method property. An exemplary WriteXML method syntax is:

obj.WriteXML fileName, xmlString

The parameters noted in the above syntax are defined as follows:

fileName	The name of the multivalue file to which to write the XML data
xmlString	The string contain the XML document.

An example of the ReadXML method is as follows:

obj.WriteXML "INVOICES","<MVData><Record Id=""00123""><Field> 11256</Field> . . . </MVData>"

The WriteXML example above writes out the INVOICES item passed to it through the XMLString parameter into the INVOICES file. The example string is abbreviated but represents the same XML Invoice 00123 record depicted above under the ReadXML method description. In keeping with the standards for a multivalue DTD content model in accordance with the exemplary embodiment depicted here, the Invoice item is written out whether or not it previously existed. The default action is to overwrite if necessary.

The adapter provides methods for exposing various system properties, including, among others, a Catalog property, a Connect Timeout property, a DataSource property, and ExtendedProperties method for holding scripted instructions for hosted systems, a Location property, a Password property, a Provider property, and a UserID property. Described below is an exemplary embodiment of such system property adapter methods. As above, the exemplary embodiment depicts, as an illustrative example, adapter methods that integrate properties, data and functionality between an XML-based Internet environment and a multivalue Legacy environment, using Visual Basic script.

The Catalog property method exposes, in the exemplary embodiment depicted here, the ADODB Connection Catalog property.

The ConnectTimeout property method, in the exemplary embodiment depicted here, exposes the ADODB Connection ConnectTimeout property. The default is 15 seconds. The syntax for the exemplary embodiment of the method is:

obj.ConnectTimeout= x

The parameters in the above ConnectTimeout property method example are defined as follows:

x	This indicates how long the driver will wait for the connection to be made before timing out and displaying an error message. The default value for this is 15 (seconds)

An example of the ConnectTimeout property method is:

obj.ConnectTimeout=30

The example Connect Timeout property method above sets the ConnectTimeout value to 30 seconds.

The DataSource property method, in the exemplary embodiment depicted here, exposes the ADODB.Connection DataSource property. An exemplary syntax of the DataSource property method is:

host[:socket]

The parameters in the above DataSource property method example are defined as follows:

host	The TCP/IP hostname or IP address of the server which contains the multivalue data you wish to access
socket	You can optionally add a colon ':' and a socket number to override the default socket number of 23 (telnet). This defaults to localhost:23.

An example of the DataSource property method is:

obj.DataSource="myserver:2023"

The DataSource property method above sets the DataSource property to point to a system with hostname myserver and connect to socket 2023.

The ExtendedProperties method, in the exemplary embodiment depicted here, is used to hold the script value for scripted instructions for connecting to, for example, UniVerse and UniData systems. A script is a series of semicolon-delimited tokens used when connecting to hosted systems. With the exception of the replacement directives, /U, /P, and /L, script directives must be placed at the beginning of a token. Tokens that do not specify a directive are sent as is. The exemplary embodiment of the invention depicted here supports the following script directives:

Directive	Meaning
/Ax	Send ASCII character code x (0..255).
/U	Replace with the value entered for User Name.
/P	Replace with the value entered for Password.
/L	Replace with the value entered for Location.

/MSs	Set the match string value to s.
/Tx	Perform a case insensitive match with the current match string. Timeout after x milliseconds (x cannot be negative).

An ExtendedProperties method example is as follows:

obj.ExtendedProperties="/MSlogin:;/T4000;liboledb startup; \L;/A13;/ MSPassword:;/T2000;/P;/A13"

In the ExtendedProperties method example above, and as depicted in FIG. 24, the invention provides the following instructions and essential information. First, the invention instructs the adapter to look for the login prompt 170. Then, the invention waits up to 4 seconds for it before proceeding 171. Next, the invention sends liboledb startup followed by the Location to which to connect and a carriage return 172. Then, the invention instructs the adapter to wait for the Password prompt 173. The invention then waits up to 2 seconds for it before proceeding 174. The invention next sends the password (from the Password property) followed by a carriage return 175.

The Location property method, in the exemplary embodiment depicted here, exposes the ADODB.Connection Location property. It is useful with respect to, for example, Unidata and UniVerse connections, to determine the identity of an account or directory in which to work. It is not typically used in situations where the user-ID implies a location. When connecting to, for example, UniVerse and Unidata systems, the Location property value returned by the Location property method is denoted by the script directive /L. An example of the Location property method is:

obj.Location="/ud33/accounts/SALES"

The Location property method example above sets the path to the VOC of the account to which the system developer will be connected, which in this example, is a Unidata structure.

The Password property method, in the exemplary embodiment depicted here, exposes the ADODB.Connection Password property. When connecting to, for example, UniVerse and Unidata systems, this value is denoted by the script directive /P. An example of the Password property method is:

obj.Password="my!really!hard$to%crack@pasword"

In the Password property method example above, the system sets the password property to a password that is, hopefully, secure.

The Provider property method, in the exemplary embodiment depicted here, exposes the ADODB.Connection Provider property and defaults to "GAXOleDB." An example of the Provider property method is:

```
obj.Provider= "GAXOleDB"
```

The Provider property method example above sets the Provider property to its default value.

The UserId property method, in the exemplary embodiment depicted here, exposes the ADODB.UserId property. On native multivalue systems, such as mv.BASE, this is the name of the account to which to logon. When connecting to, for example, UniVerse and Unidata systems, this value is denoted by the script directive /U. An example UserID property method is as follows:

```
obj.UserId="ORDERENTRY"
```

The example UserID property method depicted above logs the user/developer into an account known as the ORDERENTRY account.

The adapter of the invention further provides methods to create adapter accessible objects, for establishing connections with such objects through which the adapter can operate, and for handling error and failure conditions.

For example, the following exemplary code create an illustrative object that is accessible by the adapter:

```
set obj=Server.CreateObject("GAX.mvAdapter")
```

In order to establish an mv.BASE connection, for example, for the above-created object, the following example code is representative:

```
set obj=CreateObject("GAX.mvAdapter")
obj.ConnectTimeout="30"
obj.DataSource="10.0.0.1:2023"
obj.Location=""
obj.Script= ExtendedProperties
obj.UserId="ORDERENTRY"
obj.Password="LETMEIN"
```

In order to establish a Unidata connection, for example, for the above-created object, the following example code is representative:

```
set obj=CreateObject("GAX.mvAdapter")
obj.ConnectTimeout="30"
obj.DataSource="10.0.0.1:2023"
obj.Location="/u/data/ACCOUNTING"
ExtendedProperties="/T1;/U;/A13;/T2;/P;/A13;/T2;LOGTO
;/L;/A13"
obj.UserId="liboledb"
obj.Password="holdmeback"
```

In the previous examples, the adapter object is created in the first line of code; the ConnectTimeout, DataSource, and Location property values are specified in the second through fourth lines respectively; and the actual connection is established (using scripting directives) in line five.

The adapter provides for error and failure handling. The adapter returns notification of errors, such as ODBC, OLE DB, and ADO errors, URL connectivity errors, and other errors, to the controlling page so that the source of the error can be clearly identified. From the error message received, the source and the layer of the application in which the error occurred can be identified. The following example shows an illustration of simple error handling using VB Script:

```
on error resume next:
obj.DeleteXML "INVOICES", "<MVData><Record Id =
""00123""/></MVData>"
If Err <> 0 Then
        MsgBox Err.Description, , "For some reason, miss-
                spelled words cause problems!"
End If
```

The error-handling example code above provides for the reporting of an error that could occur when using the DeleteXML method.

When ADO connection errors occur, the adapter sets a property, such as the Err.Source property, to "Provider."

Internet Data Integration

Another aspect of the present invention provides control and integration of Internet

data from within any Windows Scripting Hosted language, including among others, Visual Basic Script (VB Script), JavaScript, and PerlScript. The data resulting from the legacy data to Internet data conversion described above is returned to a Web client in XML form. The Web client can then use (through Java, VBScript, JavaScript, PerlScript, and others) this data to enable the user to view, select, modify, combine and extract the data, or portions of the data. The way in which this aspect of the invention works is explained below.

This aspect of the invention is a language add-in which provides for easier handling of tree-based electronic representations of marked up documents, such as XML documents. The embodiment described herein for illustrative purposes uses as an example subject mark up document, an XML document.

In one embodiment, a standard XML parser is used to provide some of the required functionality. In such an embodiment, the invention exposes some of the objects of the underlying standard XML parser as static properties. In this way, the invention allows developers to access new features of new implementations of the standard XML parser, as the standard is changed. References in relation to this aspect of the invention to DOM calls refer to calls to this standard XML parser. In this way, the invention can provide access to the complete set of W3C (World Wide Web Consortium) defined objects such as the Document, Element, Attribute, NodeList, NodeMap and all other objects implemented in the standard XML parser.

One feature of this aspect of the invention is that it provides for "nodepath" notation reference to the nodes of the DOM tree representation of a particular XML document. Nodepath notation is a simplified notation for referring to XML nodes that uses a "dot-notation" format. Dot-notation is a format that is familiar to programmers using scripting languages such as Microsoft Visual Basic and other scripting languages.

The nodepath dot-notation feature of the invention provides programers with the ability to reference a child node in an XML document tree using an intuitive nodepath description of the child. For example, an XML document called "invoice.xml" contains the information as depicted in FIG. 25. As shown in FIG. 25, the information is described by the highest level start-of-data tag of "<invoice>" 801. The "<invoice>" 801 is comprised of "<invoiceNumber>" 802 with a value of "123" 803 (and an end-of-data tag "</invoiceNumber>" 804) and of "<orderData>" 805 with a value of the text string "1999-10-27" 806 (and an end-of-data tag "</orderDate>" 807). The "</invoice>" tag 808 signifies the end of the invoice data. For purposes of illustrating this feature of the invention, the executable program logic providing this feature is referred to herein as "BizDOM." Below is example code to invoke the executable BizDOM program using the VBScript scripting language to call the BizDOM feature.

```
Object obj = Server.CreateObject("Liberty.BizDOM")
```

FIG. 26 provides example VBScript code that follows the initial call invoking the BizDOM feature and which provides a nodepath reference to the orderDate text string "1999-10-27" 806. The example code as depicted in FIG. 26 shows that the invoice record is read by identifying the document as an object with "obj" 811 followed by a period "." 812 followed by the method "loadXML" 813. The loadXML method 813 is qualified by a designation of the name of the file "invoice" 816 which is followed by a period 817 and the file type "xml" 818, and which is contained within matching parenthesis 814 and 820 respectively, and further contained within matching quotes 815 and 819 respectively.

To reference the orderDate text string "1999-10-27" 806, the VBScript code identifies the data as a "String" 821 and gives the data a VBScript name of "date" 822. The actual text 806 is referenced by following the entire path from the highest level of information, "obj" 811 (followed by a period 824), to the next level of information, "invoice" 816 (followed by a period 825), to the next level of information, "orderDate" 805, and finally to the text 827 contained at that lowest node, "1999-10-27" 806.

The full reference to the text level, "obj.invoice.orderDate.text," illustrates the dot-notation nodepath syntax which is a feature of this aspect of the invention.

In order to provide the features described herein, this aspect of the invention interacts with the scripting language used by the programmer to invoke the BizDOM feature. Languages, such as scripting languages, use certain control programs, such as ActiveX controls, to provide a means to validate method and property names called by the programmer concerning a particular object. Such a validation means is sometimes referred to as a scripting language interpreter. The embodiment described herein uses, for illustrative purposes, ActiveX controls as the control program that provides the immediate interface with the example scripting language. In one embodiment, the invention is programmed using C++ code, and using Visual Basic to provide some aspects and features. In one embodiment, some aspects of the invention are implemented as an ActiveX object on top of an MS IXML DOM ActiveX Object; in such an implementation, many W3C DOM functions are delegated to the ActiveX object.

Once the scripting language interpreter knows the name is valid for the object, it then calls a different method to invoke the property or method code. In the embodiment described herein, these behaviors are supported by implementing the ActiveX IDispatch interface and its GetIDsOfNames and Invoke methods.

When the scripting language, for example, Visual Basic Script (VBScript), tries to interpret (or "parse") the reference "obj.invoice.orderDate.text", it does so one segment at a time. VBScript starts with the term "obj" 811 and searches its symbol table. In the embodiment described here, the VBScript program finds the term "obj" 811 in its symbol table as an ActiveX object. After it resolves the highest level node, in this case "obj" 811,

the scripting language relies on the ActiveX control to determine whether or not the properties and methods the programmer is calling with the subsequent nodes are valid.

The scripting language does not internally know what syntax an ActiveX control supports. Rather, the scripting language calls QueryInterface for the specified object in order to get the object's IDispatch interface. Once the scripting language has access to IDispatch, it calls IDispatch::GetIDsOfNames("invoice") to determine if the name "invoice" is a valid property for the object. The scripting language leaves the determination to the object as to whether a particular property or method name, such as the example here, "invoice," is supported as a valid property or method for the object.

At this point, it is useful to describe another feature of this aspect of the invention which is that it provides programmers with the ability to dynamically alter the tree-based representation of a marked up electronic document, such as the DOM tree for an XML document. This feature provides, for instance, the programmer to add a new node, and then immediately use a nodepath reference to the new node without any intervening program compilation step.

Objects have a pre-defined set of properties and methods. Without the present invention, if a scripting language passes a given property or method name to the object, the object can return a constant identifying the property/method or return a failure condition if the property/method does not exist.

If the object's GetIDsOfNames returns successfully, then the scripting language calls IDispatch::Invoke to do the work on the property, passing the property identifier integer and other information. The object's Invoke method has a hard-wired case statement on the property identifier and branches to code to deal with actually updating or retrieving the specified property value.

Without the present invention, if a failure condition is returned from GetIDsOfNames, then the scripting language reports an error to the programmer such as "Invalid syntax" when a property or method name has been used that is not supported by the pre-defined set of properties and methods for that object.

With the present invention, a failure condition is not returned from GetIDsOfNames. Instead, the invention builds its own version of the object's names table; the invention then adds the string "invoice" to the object's names table maintained by the invention's program if that name has not been previously added; the invention then return the appropriate index, for example, 1, of the name within the invention's name's table for that object. The scripting language then always calls Invoke with the Names table index provided, in this case, 1.

The invention provides it's own "Invoke" program code that intercepts the Invoke method sent by the scripting language. The invention's Invoke program receives the passed names table index and uses the index to look up the associated string in the object's name table maintained by the invention, "invoice" in this case. The invention's Invoke program

then uses DOM calls to look through the DOM tree for the XML document to see if there is a node at this level with the specified tag, "invoice" in this case. If found, the updating or retrieving of the property value is done using DOM calls. If not found, then IDispatch::Invoke returns a failure condition and the scripting language will report an error such as "Invalid syntax".

By searching the DOM tree each time the scripting language wants to resolve a property name, the latest additions, deletions and updates to the DOM tree are always reflected, resulting in the behavior of an object with dynamic properties - properties that appear and disappear as the programmer modifies the XML document tree.

The embodiment of this aspect of the invention resolves a property path in the following way. After calling IDispatch::GetIDsOfNames("invoice"), the scripting language then calls IDispatch::Invoke(1, get) to retrieve the "invoice" property, which in turn uses DOM calls to look through the DOM tree.

If found, the invention constructs a node object for "invoice". The scripting language then repeats the process with the returned "invoice" node object, calling QueryInterface to get the object's IDispatch interface. The scripting language then calls GetIDsOfNames("orderDate") which, with the invention, adds "orderDate" to the invention's names table if the name has not been previously added; the invention returns the appropriate index, for example, 2. Then the scripting language calls IDispatch::Invoke(2,get) to get the orderDate property which, with the invention, returns another node object, in this case, for orderDate.

The scripting language then repeats the process for the last time to get the value of the "text" property. The scripting language calls QueryInterface to get the "orderDate" node object's IDispatch interface. The scripting language then calls GetIDsOfNames("text"). The term "text" is a pre-defined term to the invention common to all node objects. The invention then returns a constant, for example 99999, to the scripting language. The scripting language, e.g., VBScript, then calls IDispatch::Invoke(99999,set, "1999-10-27"). The invention intercepts the scripting language call to Invoke. The invention's Invoke method recognizes that property 99999 is the "text" property. The invention's Invoke method then uses DOM calls to update the text for the node to "1999-10-27".

Using the invention, a programmer can alter a DOM tree by adding a new node and then immediately use a nodepath to reference the new node. Using the XML invoice document example as depicted in FIG. 25, the following line of code would cause an error because there is no tax node in the document:

```
String tax = obj.invoice.tax
```

However, using the invention, the programmer can add a new tax node by appending

a tax node using the code:

obj.invoice.append("tax").

Once the tax node is added, the following statement is valid:

String tax = obj.invoice.tax

Where a tag, such as <invoiceLine>, occurs more than once, the invention uses array references to differentiate the multiply occurring tags.

Both attributes and elements can be referred to using the same path syntax. In the rare instance that attributes and elements can have the same name, using the collection element node path distinguishes a reference from an attribute node from a reference to child element node with the same name. For example, book.author(n) is always the element, while book.author is a search node path that initially checks to see if it is an attribute and only references the element if there is no author attribute. In other words, in search node paths, attribute nodes take precedence over element nodes.

The invention provides the dynamic behavior as described above in a similar manner with regard to deletions and updates, as well as additions.

The invention provides for the creation of a new attribute in an existing element.

Collection constructs such as "for each" methods are frequently provided by scripting languages. In one embodiment, the invention exposes as a collection of nodes, groups of elements that are child elements to a specific element. In this way, the invention supports collection-accessing syntax such as "for each" methods.

Collections are a generalized method of exposing a number of objects so that a software system can iteratively process each member of the collection. According to one authority, a collection is "[a] group of objects. An object's position in the collection can change whenever a change occurs in the collection. Therefore, the position of any specific object in the collection is unpredictable. This unpredictability distinguishes a collection from an array." *Microsoft® Excel version 5.0 Visual Basic Programmer's Guide*.

In a Microsoft Visual Basic development environment, the systems developer must have a tight coupling with Microsoft Component Object Model (COM) technology In order to expose these objects, the person developing the object is required to implement a number of interfaces that are defined by Microsoft. See the definition of collections on page 65 of the guide. On page 66, there is a list of the properties and methods that a user must implement to support collections. The following example is illustrative:

<MyData><X>sunflower</X><X>zinnia</X>

<Y>cereal</Y></MyData>

In the above example, there are three child nodes under the "MyData" node. Two of them are called X, one is called Y. The invention provides for two mechanisms for exposing these child nodes as collections. One method is to find all elements with a specific name – childNodes("X") – which would return a collection of two nodes. Another is to expose all child elements at a specific level. A person with ordinary skill in the art will understand that these are two of many possible criteria that could be applied to generate a subset. Such nodes are then exposed through a standard Visual Basic Collections interface.

In the exemplary Microsoft XMLDOM/multivalue environment embodiment, exposing nodes through a standard Visual Basic Collections interface is accomplished by creating a class to which reference is made herein as "mvCollection". The mvCollection class exposes a number of properties and methods concerning the referenced collections, and can be initialized from underlying DOM objects.

The mvCollection properties and methods provided by the exemplary Microsoft XMLDOM/multivalue environment embodiment conform to the rules governing a standard OLE/VB collection. As such, the mvCollection class has methods and properties that have to be supported by the collection class according to OLE standards. For mv.DOM the following standard properties and methods for a collection object are supported:

New Enum	Returns an OLE object that supports IEnumVARIANT. This method can not be accessed by users.
Item (n)	Returns the indicated item in the collection, or VT_EMPTY if the item does not exist.
Count	Returns the number of items in the collection.
Add	Adds the indicated item to the collection.
Remove	Removes the specified item from the collection.

Collections return their count, as well as standard ActiveX enumeration. As a collection is updated, new elements are immediately reflected in the count and enumeration.

In the exemplary Microsoft XMLDOM/multivalue environment embodiment, collections can contain just a single element or multiple elements. For example, document.author and document.author(1) are both valid collections even if author is defined only as a single element. Similarly, the first element of a multiple-valued collection can be referenced as myDoc.Order.detailLine as well as myDoc.Order.detailLine(1) even if the collection is defined to have more than 1 element.

In the exemplary Microsoft XMLDOM/multivalue environment embodiment, Node Paths can be used to reference a collection of elements. Modern collections in VBScript and JScript are one-based. That is, every element has a collection of child elements even if that

collection will only consist of one element. The following example is illustrative of referencing a collection of elements using Node Path notation:

```
collectionelementnodepath =
    nodepath.collectionelementname (index)
```

Following is an explanation of the parameters appearing in the above collection reference example:

nodepath	This identifies the path of the parent node of the collection in the XML document tree. If the nodepath results in an attribute instead of an element, an error is thrown.
collectionelementname	This identifies the element name shared by a set of child nodes.
index	This is a one-based integer identifying particular child nodes within a collection on a left-to-right basis. However, collection elements are only treated logically if they are adjacent If other child nodes not matching the specified collectionelementname are interspersed, they may be ignored.

In the exemplary Microsoft XMLDOM/multivalue environment embodiment, performing the procedure as depicted in FIG. 27 and as described as follows retrieves a collection of mvNodes. First, start Visual Basic 177. Then, create a Standard EXE project 178. Next, add a button to the form 179. Then, go into 'P'roject, Refere'n'ces and check GA eXpress mvDOM 1.0 Type Library 180 (Note that the names of the files and libraries are exemplary for illustrative purposes and are not a limitation of the invention). Then, click "OK" to the dialog box 181, double click the button, so the code window for Method1_Click is displayed 182, and enter collection retrieval code 183, such as the code depicted in FIG. 28, in the window provided.

In the above example, the first section of the collection retrieval code depicted in FIG. 28 400 - 408 creates an mvDOM object and loads it with some data as follows:

```
Dim objmvd As New GAXLib.mvDOM
objmvd.createRoot("MVData").append("Record")
appendAttribute("Id") ="abc"
objmvd.MVData.append("Record").appendAttribute("Id")          = "def"
objmvd.MVData.append("Record").appendAttribute("Id")
= "ghi"
```

```
objmvd.MVData.append("foo") = "bar"
```

The tree-view representation of this data is represented as follows:

```
<MVData>
<Record Id="abc"/>
<Record Id="def"/>
<Record Id="ghi"/>
    <foo>bar</foo>
</MVData>
```

The next section of the collection retrieval code depicted in FIG. 28 409 - 411 and as depicted below specifies a For-Each loop to go through all child nodes of the MVData element causing a display of a dialog box with the name of each child node. The Record would be displayed three times followed by a single display of "foo".

```
For Each RecItem In objmvd.MVData.childNodes()
  MsgBox RecItem.Name, , _
    "For Each with childNodes()"
Next
```

The next section of the collection retrieval code depicted in FIG. 28 412 - 414 and as depicted below uses a For-Each loop that gets all child nodes of the MVData element that are called Record causing the display of a dialog box three times, showing the name Record.

```
For Each RecItem In objmvd.MVData.childNodes("Record")
  MsgBox RecItem.Name, , _
    "For Each with childNodes(""Record"")"
Next
```

Lastly, the next section of the collection retrieval code depicted in FIG. 28 415 - 420 and as depicted below explicitly retrieves the Dim objmvColl As New GAXLib.mvCollection, uses a simple for-next loop with a count of childNodes to create a collection of mvNodes,

and then successively accesses each Item by looping through the collection of mvNodes using an index with the standard collection property, "Item".

```
    Dim objmvColl As New GAXLib.mvCollection.
    Set objmvColl = objmvd.MVData.childNodes()
    For i = 1 To objmvColl.Count
  MsgBox CStr(i) & "=" & objmvColl.Item(i).Name, , _
    "Looping through using Item(n)"
Next
    End Sub
```

Internet Data to Internet Data Integration

The "meta language" capabilities of XML that provide for the explicit declaration by each development programmer of new DTDS has resulted in each Internet business independently in many cases developing a proprietary XML structure and associated set of naming conventions. In response to the diversity of formats, the present invention provides apparatus, systems and methods for one Internet application to use XML documents from multiple enterprises.

From the legacy data to Internet data conversion described above, an XML object is available to the browser that contains the legacy data, but it is in a form that, although modified, is reflective of the needs of the legacy DBMS and the accessing browser. Trading partners have structures and naming tags proprietary to their respective enterprises, or in the alternative, conform to one of the many available "standard" formats promulgated by numerous vendors.

The present invention provides a method, using a computer, for converting Internet data from one Internet data structure to another Internet data structure. The method is similar to the method described above for converting Internet data to a simple legacy data structure. The first Internet data structure is characterized by a tree-based structure definition. The second Internet data structure is characterized by a second tree-based structure definition.

The method for converting Internet data from one Internet data structure to another Internet data structure comprises several steps. The first tree-based structure definition comprises a first plurality of node definitions. The second tree-based structure comprises a second plurality of nodes.

The computer compares the first Internet data tree-based structure with the second tree-based structure. For each node in the first Internet data tree-based structure, the computer identifies the equivalent node in the second tree-based structure. The computer builds a translation table of the equivalent first Internet data tree nodes and the corresponding second Internet data tree nodes. The computer uses the translation table to map the first Internet data to the second Internet data tree-based structure.

Once an XML document is transformed to the desired format, it is then posted to the appropriate trading partner's Web site. The trading partner then responds with the information requested and is posted to the originating Web site. The received information can then be converted by the originating server to a form suitable for the legacy DBMS. And finally, the converted information is written back to the data base.

Legacy Data Internet Portal

FIG. 30 illustrates an exemplary embodiment of an legacy data Internet portal embodiment of the present invention in which three tiers of systems interface, the three tiers being: 1) one or more Web clients 301a-301n; 2) a Web server system 302; and 3) one or more Database Management System Server Systems (DBMS server) 305a - 305n. The use of suffixes "a" through "n" are used to depict an unlimited number of the subject element and are not a limitation of the invention.

In this embodiment, a standard architecture, such as the Microsoft Distributed interNet Archtecture (DNA), is used. The three-tiered system architecture is illustrative and not a limitation of the invention.

Continuing with the illustrative three-tier system embodiment, the client system 301a-301n tier is the tier from which a request originates. As depicted in FIG. 30, the client system tier is the top tier of the three-tier system relationship. A client is, for example, a Web browser application or a server-based application mimicking the behavior of a client application. A client may also be a page or other system component running on the Web server or the DBMS server.

As depicted in FIG. 30, Web server 302 is the middle-tier system in the three-tier system architecture. In this embodiment, the Web server provides software that applies appropriate business logic for all transactions occurring on the portal. The Web server communicates with one or more DBMS servers 305a-305n.

A DBMS server, e.g., 305a, is the server on which legacy data resides. In addition to the legacy data, each DBMS server, e.g., 305a, is also the server on which legacy software methods and programs reside. In this embodiment, the legacy software methods and programs, when executed, provide information about the particular DBMS server system 305a-305n, or about the legacy data 306a-306n residing on the respective server.

References in the description of this embodiment to "the client", "the Web Server" and/or "the DBMS server" performing an action are understood to mean that the respective server, client, Web or DBMS server, is programmed to act in the manner described.

In this embodiment, a client, e.g., 301a, originates a request for information and sends the request to the Web server. The Web server 302 receives the request. It interrogates a table 303 that contains, among other things, information with which the Web server 302 identifies the appropriate DBMS server or Servers to which the request is directed. A single Client request may require that information be compiled from several DBMS servers. Accordingly, the Web server directs the request to each of the applicable DBMS servers. For purposes of the present example, only a single DBMS server is described.

The Web server 302 maintains a table 303 that contains, among other things, an identification of the appropriate communication information that must be used in order to communicate with each of the DBMS servers 305a-305n with which the Web server 302may be called upon to communicate. For a particular request, the Web server 302 identifies the communication mechanism in the table 303 for each DBMS server to which it must direct the request or some portion thereof. The Web server 302 then formulates a command and directs a call to each of the appropriate DBMS servers 305a-305n as the case may be. The Web server 302 formulates the command and directs the call in such a manner that it is interpreted by the receiving DBMS server, e.g., 305a, as a request to execute a command or program on the particular DBMS server.

When the appropriate DBMS server, e.g., 305a, receives the request to execute a command or program known to it, the DBMS server 305a executes the appropriate command or program. The DBMS server 305a prepares as output of the executed method or program the requested information. The DBMS server 305a directs the output information to the Web server 302 for analysis.

The Web server 302 receives the DBMS server output. The Web server maintains information, for instance as part of table 303, instructing the Web server 302 as to the appropriate analysis procedure for each DBMS server 305a-305n to which the Web server communicates. The Web server 302 accesses the table 303 and determines the appropriate procedure with which to analyze the DBMS server response. The Web server 302 analyzes the DBMS server output according to the instructions retrieved from the table 303, for instance, by requesting the DBMS server 305a for the appropriate metadata and/or parsing the implicit metadata available in the DBMS server output. The metadata determined includes the number of columns of output to be generated and a mechanism for identifying and retrieving rows of data. The Web server 302 then retrieves the data. Either the Web server 302 or the DBMS server 305a ensures that the first column of data retrieved is a unique identifier within the result data set. In an alternative embodiment, the Web server

302 looks for and detects a single, unique identifier within the result data set, for instance, a unique row number.

When the Web server 302 is ready to retrieve the data, the Web server creates storage space for the result data set according to a tree-based Internet data structure 304, such as, for instance, an XML document. The Web server 302 initializes the storage space with an empty root element named, e.g., "Records". For each record that the Web server 302 retrieves, the Web server 302 creates an empty record element as a child element of the Records root element. The Web server 302 then reads the unique identifier element described above and saves the content of the unique identifier element as the content of the data portion of the Internet data structure attribute, e.g., for instance, an XML attribute, for the new Record element that it previously created. For instance, the Web server 302 would name the unique identifier as "ID" and set the "ID" attribute to the contents of the unique identifier element.

For each column of the output data generated by the executed method or program, the Web server 302 adds a new empty "Field" element as a child element of the Record element. The Web server 302 then reads the data for each column and adds the data as a new text node to the "Field" element that it previously added. The Internet data structure 304, e.g., the XML object, is complete when the Web server has completed reading all of the columns for all of the rows for the output data set.

It will be understood by someone with reasonable skill in the art that the description above of certain name tags is exemplary and not a limitation of the invention. In addition, the identification of a single "Records" root element is exemplary and is not a limitation of the invention. To the extent that the DBMS server provides enabling metadata, the Web server can identify and use appropriate naming tags.

An illustrative example is provided involving the development of a Web site which will in turn communicate with a number of DBMS servers. In the example, a web site is being developed which will allow an administrator to query the number of users running on a number of DBMS servers via a Web browser. In the example, it is desired that the system periodically generate statistics about the usage of the various DBMS servers by various users. In order to provide the maximum leverage of the legacy program code and at the same time, use the latest technology, the developers of this application use an XML technology embodiment of the legacy data to Internet portal.

The particular example provided illustrates the capability of the legacy data to Internet portal to expose information captured by the DBMS server that is not typically contained in the database stored on the DBMS server. Information about the state of the DBMS server machine, including such information as the number of users running on the DBMS server machine is not typically contained in a database. For example, in SQL Server 6.0, there is no table or database which contains data tracking the number of users connected

to the SQL Server. However, executing a stored procedure, 'sp_who', on such an SQL server will provide a list of currently connected users.

In contrast the example SQL server described above, some multivalue systems provide a table which contains data that, among other things, implicitly indicates the number of users connected to the system. In order to interpret the implicit data, the multivalue DBMS vendor provides a method, 'LISTU', that applies software to provide more information than is available in the database tables.

In both of the cases described above, programs or methods can be executed on the appropriate DBMS server to generate the desired information which is not explicitly contained in the raw data in the databases provided on those servers.

In the example case of the example multivalue DBMS server system, a method-line mode is provided called Terminal Control Language (TCL) from which programs and methods can be executed. In the example case of the example multivalue DBMS server system, the 'LISTU' method is provided. FIG. 31 depicts an exemplary 'LISTU' method 311 and an exemplary 'LISTU' output 312 result as displayed on an ASCII terminal.

FIG. 32 depicts an exemplary legacy data Internet portal method using Visual Basic Script and XML. As depicted in FIG. 32, the language parameter is set to "VBScript" 350. The start 351-1 and end 351-2 XML active server page code delimiters mark the beginning and ending of a scripting fragment comprised of a series of Visual Basic methods that include references to a specific implementation of an XML object. The "set obj" method 352 instructs a Web server to use the "CreateObject" method. The DataSource 353, UserId 354 and Password 355 properties are set to allow the object 'obj' to connect to the appropriate DBMS server. The readXML method 356 sends the method 'LISTU (N' to the DBMS server for processing. The last two methods 356 and 357 format the resulting ASCII XML data with appropriate headers to be transported to the client browser and identified as being of type 'text/xml'.

Note that both the first character ">" 310 is a TCL prompt character, that 'LISTU (N' is the command as issued and that everything below it is the output 312 of the method.

In the example, a browser application has been created that every 60 seconds requests an XML document to be generated by a Web server. The Web server then determines that it must query several multivalue systems for the number of users connected. For each multivalue system that it must query, the Web server does the following: 1) identifies the DBMS server (the multivalue system) that it must communicate with; 2) determines what type of connection is being made, in this system it is an ADO connection using an OLE DB driver; 3) looks up the type of connection and determines that there is a mechanism to pass DBMS server methods to the DBMS server; and 4) formulates an appropriate method and sends the method in the indicated manner to the appropriate DBMS server.

The DBMS server executes the method in such a manner that the output is captured as a variable or other parameter that is returned back to the Web server.

The Web server determines from the information contained in its table of DBMS server communications that: 1) each line of output 312-317 as depicted in FIG. 31, is interpreted as a single row comprising a single column of data (there are six lines of data, therefore there are six rows, each row comprising a single column of text); 2) a unique identifier for each row is created by inverting the row numbers, and pre-pending the identifier as an element to each row -- in this way the last row is identified as row '1', the first row is identified as row '6'-- this identifier then becomes a new column of data in the output of each row.

FIG. 33 depicts an exemplary embodiment of an XML tree-based data structure built by the Web server through the Legacy data Internet portal using the DBMS output data depicted in FIG. 31. The Web server creates an empty "<Records>" root element 330-1 and 330-2 as soon as it knows that it can retrieve the result set. For each row read in, the Web server creates an empty record child element within the records element (each empty record child element comprises the "<Record>" and "</Record>" start and ending tags, e.g., 331-1 and 331-2; 334-1 and 334-2).

The Web server parses each row to determine it's implicit metadata. It finds the unique identifier, in this case the first of the two columns, and reads the data in that column. This data is used to create an "Id" attribute, e.g., 332, (id="6") and 335 (id="3") of type ID in the record element for this row. The Web server then reads in all remaining columns of data (in this example, there is only one remaining column of data, e.g. 337) and for each, creates a rield element, e.g. 336-1 and 336-2 as a child element of the Record element. The Web server reads the data for each column, e.g., 337 as depicted in FIG. 31, and places it as a text node 337 as depicted in FIG. 33, within the field element, e.g. 336-1 and 336-1 that was created to hold it. When all columns and rows have been read in, the XML document is complete and the result set is closed. In an alternative embodiment, the Web server may do additional parsing.

Having extracted the necessary data, the Web server 302 communicates the information back to the client, e.g., 301a. The client, e.g., 301a, in turn performs calculations on the data that it receives and then formats the output from all the servers that it has queried to the browser.

In an alternative embodiment, a program is created on the multivalue server which generates additional columns of output that identify, e.g., the user id (LIBERTY, JOHN and TIM) for each user that was signed on to the system at the time the inquiry is made, the time and date at which each user first logged on, and, as appropriate, some calculations on the information.

Additional Features

A name collision situation arises in the circumstances where a static Interface Definition Library (IDL) defines a name, and the XML object with which interface is required uses the same name. The XML object could reference, for instance, the same name both as an attribute and as an element. A collision situation also arises in the circumstances where Visual Basic defines a term as a keyword and enforces capitalization of that term whenever it is used. In summary, the sources of naming issues are: static names (standard and special properties and methods), element names, attribute names, and keywords. The present invention provides a collision resolution.

XML allows free-format naming of elements and attributes, and provides some of it's own names for standard objects that can exist within a DOM object. One example is the #text object, which is the name given to a standard text node in a DOM document. XML does not restrict naming beyond a certain set of characters that cannot be used in an element or attribute name.

In an object-oriented world, there are inevitably base properties and methods that an object exposes to allow standard operations to be performed on the object's data. These are referred to as static identifiers and are generally exposed in a type library (TLB) or IDL.

The invention, as illustrated in the exemplary XMLDOM/multivalue embodiment described further below, exposes XML elements and attributes as properties. The invention provides a programmer collision-resistant access of attributes and elements. The invention provides, in one embodiment, for the use of an ActiveX control in an intuitive way by Visual Basic programmers to expose the elements and attributes of an XML object as properties. This is accomplished in a way that is natural and intuitive to a Visual Basic programmer, and does not require such programmers to change subtle settings in the Microsoft Visual Studio Integrated Development environment (IDE). The manner in which collisions are handled in the exemplary XMLDOM/multivalue embodiment is summarized below.

access to:	Use of the following will resolve a collision issue:
an element	A collection index or element(tagName) For example, customer.name(1) or customer.element("name") accesses the first child element with tag name of customer.
an attribute	An attribute(tagName) For example, customer.attribute(name) accesses the attribute of customer with tag name.

	(a child element with tag = "name" should not be used)
A built-in property or method	The built-in property or method For example, customer.name accesses the name property (customer) of the customer element.three. There are three special static names 'name', 'count', and 'text' which are referred to as "special static names"; all other names are referred to as "standard static names".

In the exemplary XMLDOM/multivalue embodiment, when the mv.DOM object parses a name in a nodepath, the logic functions depicted in FIG. 29 are performed as described below:

1. The invention first checks a name in a nodepath against all existing methods and properties 421a-421b, with exception of three properties— name, count, text. The properties and methods checked, other than name, count, text, are standard static names from the IDL.

2. If the name is recognized as one of the existing methods (that is, one of the standard static names), it is invoked 421c.

3. If the name parameter/argument matches a standard static name, then the parameter/argument must match the method or property 421d. If the arguments are valid, the method/property is called 421f. Otherwise, if the arguments are not valid according to the named method or property, an error occurs 421e. Consider the following example:

MyObj.Invoice.append("gfgfg")

The above example would correctly call an existing method. In contrast, the following example illustrates an incorrect approach that would result in an error condition:

MyObj.operations.append = "append operator".

4. If the name is not one of the existing names (that is, one of the standard static names), then the parser determines whether there is an index422a attached to the

name. If the name has an index, then the name is assumed to be an element name of the underlying XML object being represented 422b. The parser assumes that it is an element name and the nodepath refers to the appropriately numbered element with the specified name. The following is illustrative of a correct example:

MyObj.Invoice.Details.Line(7)

In the above example, the seventh element with the name Line is retrieved. In contrast, the following example illustrates a correct syntax that nevertheless results in an error because there is only one element called Invoice.

MyObj.Invoice(2)

5. If there is no index in the nodepath, the parser will compare the passed name with the predefined, existing standard properties 423a —name, count, text. That is, in the absence of an index, the invention checks to see if the name is one of 'name', 'count' or 'text' (one of the "special" static names). If one of the "special" static names is found, then the corresponding standard property will be invoked 423b.

6. If there is no index in the nodepath, and the string is not one of the pre-defined, existing names, the parser first tries to find an attribute with the specified name 424a. The parser checks to see whether an attribute has been found 424b. If an attribute is found, it is retrieved 424c.

7. If the attribute is not found, the parser will try to find the child element with this name 425a. The parser checks to see whether a child element is found 425b. If so, the child element is retrieved 425d. Otherwise, if neither an attribute nor an element are found, an error message is displayed 425c.

8. If an element has an attribute and a child element with the same name, to specify the element in the nodepath use index or the element method.

The invention also provides an "alias" instruction. The following example syntax is exemplary:

alias("Field","field")

The alias instruction is provided because XML element and attribute names are case sensitive, and further because Visual Basic forces keywords to upper case. Accordingly, the above exemplary alias instruction syntax is provided to allow us to see the string 'Field' as the value 'field'. Without the alias instruction, by default, the instruction "obj.field" is interpreted by Visual Basic as 'field' being a keyword; Visual Basic then forces the word "field" to be expressed in the capital case as "Field". The alias method instructs the parser to see 'Field' and know that it must really look for and deal with an element called 'field'.

In addition, the invention provides further instructions, including 'attributeName("xxxx")' and 'elementName("xxxx")'. the attributeName and elementName instructions force the parser (the term "parser" is used herein to mean a program, a program routine, a subroutine, a computer programmed in such a way, or the like, as to accomplish the features described) to find an attribute or element by the appropriate name and return it's Node object.

For purposes of illustrating collision handling features in the exemplary XMLDOM/multivalue embodiment, the following examples are provided.

An exemplary DTD defines that the element Employee has an attribute called name and could contain a child element called name as well. The following example is illustrative of a correct line of code to return the number of siblings of the Line element which have the same name:

```
MyObj.Invoice.Details.Line.count
```

The following example illustrates a correctly coded instruction to return the number of siblings with the name Line after the 7-th element (7-th element included).

```
MyObj.Invoice.Details.Line(7).count
```

The following example illustrates an incorrectly coded instruction:

```
MyObj.Department.Employee(3).name.FirstName
```

In the above example, the "name" without index is treated as a standard property and an error results.

These following three examples are correctly coded to each retrieve the attribute name of the third element with the name Employee:

```
MyObj.Department.Employee(3).name(1).FirstName
MyObj.Department.Employee(3).element("name").
```

FirstName

MyObj.Department.Employee(3).attribute("name")

The invention provides a "text" property that preserves the value of a particular attribute with all white spaces preserved. In the exemplary XMLDOM/multivalue embodiment, the mvNode text property is different from the Microsoft IXMLDOMNode text property. The mvNode text property represents, for an attribute, the value of the attribute with all white spaces preserved; for an element, the mvNode text property represents the value of the first text node that is a child of the subject element. The following example illustrates the mv.DOM syntax for accessing as a text child of a particular element:

MyObj.Invoice.Details.[#text](j)

In the above example, if "j" has a numeric value, the above example method will retrieve the j-th text child of the Details element.

The invention provides for the retention of all white spaces. In the exemplary XMLDOM/multivalue embodiment, white spaces are defined as newline, tab, and space characters. In desktop publishing style markup, white space is viewed as extraneous. On the other hand, in a multivalue database environment, white spaces are viewed as crucial to the integrity of the data. Unlike HTML which ignores white space, XML has the capability (through the xml: space attribute) to retain all white space. The mv.DOM text property employs the following strategies when handling white spaces:

1.) It always keeps white spaces exactly as they were in the source document. That means that all significant white spaces, spaces inside attribute values and inside the text content, are preserved.

2.) It may or may not create an extra text node for insignificant white spaces—the white spaces between tags. The behavior is determined by the current value of the xml:space attribute of the enclosing element.

3.) It eliminates insignificant white space if the xml space attribute does not exist or its value has been set to default,

4.) It preserves insignificant white space by creating additional text nodes if its value is set to "preserve."

The invention further provides a browser interface for managing "Q-Pointers", such as in a multivalue system environment. A Q-Pointer is a synonym reference to a file, where

the file being referenced can be in the same account, but under a different name, or in a different account with either the same name or a different name.

In order to provide a browser Q-Pointer interface, an exemplary embodiment of the invention uses a particular routine containing information about the machine and the account to which to connect for access of a particular document. The name of such a routine in an exemplary Internet environment is the "global.asa" routine. Most other pages use the global.asa routine information to access an account.

Using the global routine allows easy customization of the Q-Pointer interface for different environments. However, someone with ordinary skill in the art will understand that use of the global routing is not a limitation of the invention. For example, in an alternative embodiment, instead of using global values obtained from a global routine, the Q-Pointer interface uses hard-coded values for the interface information.

The first web page (default.asp) of the invention accesses the account identified by the global.asa routing, and retrieves a list of Q-Pointers with their relevant information. The invention then presents the list of Q-Pointer as a browser readable page for display on the user's display monitor. In an exemplary embodiment of the browser interface, as depicted in FIGS. 21a-21b, each Q-Pointer is displayed along with options to add new Q-Pointers or delete any existing ones.

The invention uses another web page, "DeleteQPointer.asp" to delete a Q-Pointer from the account and return a browser page that indicates the status of the requested operation and returns operation back to the first web page.

The invention uses another web page, "NewQPointer.asp", to add a new Q-Pointer. NewQPointer.asp presents a simple HTML form through which the invention requests and retrieves information for a new Q-Pointer entered by a user. The invention then submits the user-supplied new Q-Pointer information to another page, "AddQPointer.asp". The AddQPointer.asp page accepts the HTML form values from the previous NewQPointer.asp page, parses this information, and uses it to validate the creation of a new Q-Pointer that conforms to the user-supplied information. If the addition of the new Q-Pointer is validated, then the AddQPointer.asp page Creates the new Q-Pointer and returns a browser page which displays the status (success or failure) of the operation as well as an option to go back to the first page.

As used herein, the term "page" refers to, among other things, an Active Server Page, a servlet, ISAPI, NSAPI or other extension technology. Someone with ordinary skill in the art will understand that the specific use of Active Server Pages in the exemplary embodiment described here is illustrative and is not a limitation of the invention.

As used herein, the term "Browser Page" refers to the output of the server page. The server page generates output which is read by a browser in response to the POST or GET request that the browser sent to the specific server page. The browser interprets this page and exposes it to a user as a user-interface.

An exemplary embodiment of the Q-Pointer browser interface logic is depicted in FIG. 34 and an example is provided below. In the exemplary embodiment, the Q-Pointer browser interface comprises a series of sample web pages that expose an account's Q-Pointers, and provide for the addition of new Q-Pointers and/or the deletion of existing Q-Pointers from an account. In the example provided below, a Q-Pointer called RHQ is added and deleted. The invention, in the exemplary embodiment depicted here, provides the Visual Basic code described below in a way that is invisible to the user. In general, the interface logic as depicted in FIG. 34 is to establish a connection 430; set the connection properties 431; get the itemId, FileName, and AccountName using query strings 432; ensure that nothing will be overwritten 433; and build the item 434.

Exemplary Visual Basic code for establishing a connection and setting connection properties is as follows:

```
set obj=Server.CreateObject("GAX.mvAdapter")
obj.DataSource=Application.Contents("OLEDBDataSource")
obj.UserId=Application.Contents("OLEDBUser")
obj.Password=Application.Contents("OLEDBPassword")
```

The above lines of code provide a link with which to establish a connection. The user clicks on the link to invoke the access to the server. The server is accessed through a mechanism called resource pooling. The exemplary embodiment of the invention depicted here uses Microsoft's Distributed Internet Architecture (DNA) to achieve high scalability.

Once the link is open and the connection is made, an interactive list of available Q-Pointers 440, accounts 441, and files 442 is displayed as depicted in FIG. 35.

Exemplary Visual Basic code for getting account, item, and file information via query strings is as follows:

```
ItemId=Request.QueryString("ItemId")
if ItemId = ""then
        ' Was it a POST request?
        ItemId=Request.Form.Item("ItemId")
end if
if ItemId = ""”then
        ' Not passed in, report this failure
        Response.Status=""
        Response.Write("")
        Response.End
end if
FileName=Request.QueryString("FileName")
```

```
if FileName = "" then
' Was it a POST request?
    FileName=Request.Form.Item("FileName")
end if
```

The lines of Visual Basic code illustrated above provide a graphic user interface with which to determine the existence of a particular item, such as the RHQ item 446 as depicted in FIG. 36, and which exposes the account, item and file information.

As depicted in FIG. 36, scrolling through the available item list 447a-447b, RHQ is an item 446 on the available item-list 447a-447b. Associated with each Q-Pointer is a graphic "button", e.g. 448, that, if clicked by the user, causes the deletion of that Q-Pointer.

Exemplary Visual Basic code for performing the precautionary step of ensuring that an existing item will not be overwritten is as follows:

```
FilterText="SELECT MD '" & ItemId & "'"
xml=obj.ReadXML("MD",FilterText,1)
set objQ=Server.CreateObject("")
objQ.loadXML(xml)
if objQ.MVData.exists("Record") then
    you cannot overwrite this item
    Response.End
end if
```

The lines of exemplary Visual Basic code illustrated above provide graphic user interface features as depicted in FIG.36 and as described below for following the described procedure:

1.) Click the Delete button 448 to the right of the RHQ item 446. A message appears reporting that it has been deleted. If an attempt is made to create a new Q-Pointer without deleting another of the same name, a message appears indicating that it already exists. Click the back button and proceed with the next step.

2.) If RHQ doesn't exist, just continue with the next step.

Exemplary Visual Basic code for building a Q-Pointer is as follows:

```
set objQ=Server.CreateObject("GAX.mvAdapter")
objQ.createRoot("MVData").append("Record").
appendAttribute("Id")=ItemId
objQ.MVData.Record.append("Field")="Q"
objQ.MVData.Record.append("Field")=AccountName
```

```
objQ.MVData.Record.append("Field")=FileName
obj.WriteXML "MD",objQ.xml
```

The above-illustrated lines of exemplary Visual Basic code provide a graphic user interface with which to capture user-entered information for building a Q-Pointer. As depicted in FIG. 35, the user clicks a Create New Q-Pointer button 443 to create the RHQ item in the Master Dictionary. A dialog box as depicted in FIG. 43 will then appear. The user then enters the appropriate information in the fields of the dialog box as follows:

Enter Q-Pointer Name 460:	RHQ 461
Enter Account Name 462:	ORDERENTRY 463
Enter File Name 464:	CUSTOMERS 465

The user then clicks on the Write This Q-Pointer button 468. A page will then appear indicating that RHQ was added 466 as depicted in FIG. 38. The user then clicks the Go Back button 467. RHQ 446 now appears in the list of items as depicted in FIG. 36. To delete the RHQ item 446, the user clicks the Delete button 448 associated with the item that the user wishes to delete.

The invention further provides a GetFile method that exposes the ability to access hierarchical data as XML through a standard HTML GET or POST request. In the exemplary XMLDOM/multivalue embodiment described here, the GetFile aspect of the invention uses a web server's programming API environment to allow it to retrieve the body of a POST request, or the QueryString element of a GET request. The content of these values are exposed either as an XML object, represented as a string, or as HTML form variables. These variables or specific tags in the XML, are extracted to provide the arguments to the methods exposed in the embodiment of the prior invention. Specifically, the FileName and FilterText values of the XML Pass-through mechanism are extracted and called. The resulting XML object is then returned as a standard http response with content mime-type 'text/xml'.

The following example is provided to illustrate retrieving data from the RHQ Q-Pointer file created in an example above. In this example, the RHQ Q-Pointer file gets its data from the CUSTOMERS file on the ORDERENTRY account. The logic for retrieving data from a Q-Pointer file as depicted in FIG. 39 is summarized as follows (the "Retrieval Logic"): establish a connection 470; set the connection properties 471; put a governor (maximum limit) on the request 472; specify the file to be retrieved using query strings 473; specify the type of request (POST or GET) 474; specify the filter text using an optional TCL statement to activate a SELECT-LIST 475; read the data 476; tell the browser it is going to receive XML output 477; send the data 478; and explicitly end the output (optional) 479.

The Visual Basic code for establishing a connection and setting connection properties as described in Retrieval Logic functions 470 and 471 is as follows:

```
set obj=Server.CreateObject(GAX.mvAdapter")
Set connection properties obj.DataSource=Application.
Contents("OLEDBDataSource") obj.UserId=Application.
Contents("OLEDBUser")
obj.Password=Application. Contents("OLEDBPassword")
```

The above lines of code provide a browser user interface, an exemplary embodiment of which is depicted in FIG. 40. Using the exemplary browser user interface depicted in FIG. 40, the user enters a URL address for GetFile.asp and presses "Enter" on a user input device, such as a keyboard, to open the page. Note that the exemplary full URL depicted in FIG. 41 is http://rhacer.libertyodbc.com/Demo/GetFile.asp?FileName=RHQ&Filter-Text=SELECT%20RHQ%20'1' which, because of the size limitation of the graphic user interface window 481 as depicted in FIG. 40, is not displayed completely. Note further that everything preceding the question mark ("?" 483) 482 in the full URL is the URL of the page that will serve the request; everything following the question mark ("?" 483) 484 is the querystring value, made up of name=value pairs, separated by ampersands ("&"). Also note that the value portion of the name=value pairs in the querystring is URL encoded as per the HTML specification for a GET method request.

Once the connection has been established quickly and easily using the invention, many of the above-described procedure steps are performed automatically. For example, the Retrieval Logic functions 472 - 475 described above involved in Specifying the Nature of the Request, require Visual Basic code such as the exemplary code ("Request Specification Code") depicted in FIG. 42 and as illustrated below:

```
maxItems=100
FileName=Request.QueryString("FileName")
If FileName = "" Then
        FileName=Request.Form.Item("FileName")
End if
FilterText=Request.QueryString("FilterText")
if FilterText = "" then
        FilterText=Request.Form.Item("FilterText")
end if
```

As depicted in FIG. 42, the first line of the Request Specification code 486 sets a governor (a maximum) that ensures that if more than 100 records are requested, only the first 100 are returned. The second through fourth lines of the Request Specification code 487-489 retrieve the FileName variable regardless of whether the FileName is provided through a GET method or POST method. The fifth through seventh lines of the Request Specification code 490-492 retrieve the FilterText value regardless of whether the FilterText value is provided through a GET method or POST method.

The lines of code illustrated in FIG. 42 and described above provide a dialog box 493 as depicted in FIG. 43, which prompts the user specify a FileName 494 and the FilterText 496. For example, the FileName entry 495 and the FilterText entry 497 are made as follows: FileName=RHQ 495; FilterText=SELECT RHQ '1 497. The corresponding filter text is generated behind the scenes.

In order to accomplish Retrieval Logic functions 476-479 for Retrieving the Data, the exemplary Visual Basic code depicted in FIG. 44 and as depicted below is illustrative of the coding required ("Retrieval Code"):

```
xml=obj.ReadXML(FileName,FilterText,maxItems)
Response.ContentType="text/xml"
Response.Write(xml)
Response.end
```

In the first line of the exemplary Retrieval Code 500 depicted in FIG. 44 and as illustrated above, the ReadXML method returns an XML object from the data source. The ReadXML method uses Resource Pooling to ensure that the best performance and scalability is achieved by the product.

In the second line of the exemplary Retrieval Code 501 depicted in FIG. 44 and as illustrated above, the browser is told to receive XML output. In the third line of the exemplary Retrieval Code 502 as depicted in FIG. 44 and as illustrated above, the data is sent. The WriteXML(xml) method specifies the XML input: the XML to write and the name of the multivalue file to write to. In the fourth line of the exemplary Retrieval Code 503 as depicted in FIG. 44 and as illustrated above, the query is completed.

The results of the Retrieval Code depicted in FIG. 44 and as illustrated above is that the first record, <Record Id="1"> is retrieved from the RHQ file (which gets its data from the CUSTOMERS file on the ORDERENTRY account). The information for R&B Auto Repair is the only record that matches the FilterText criteria (this is the item-id returned by the SELECT statement). The information for R&B Auto Repair is then displayed in the browser 485 as shown in Figure 24a.

The invention provides a GetItem extension to the above-described GetFile feature of the invention. Using the GetItem feature of the invention, instead of passing a FilterText value for display through the browser, the Item-Id value of an item, which in the exemplary embodiment is a multivalue item, is retrieved. The invention converts the item-id to an equivalent FilterText value that then enables retrieval of that item if it exists. The following example is illustrative. The GetItem feature exposes a page called GetItem.asp that the GetItem feature reads for the following variables:

FileName	RHQ
ItemId	1

The GetItem feature uses the FileName (RHQ) and the ItemID (1) to return the item with item-Id 1 from the RHQ file (which gets its data from the CUSTOMERS file on the ORDERENTRY account). This data is returned as XML and is displayed using the user's browser 511 as depicted in FIG. 45. As depicted in FIG. 45, the data for Item 1 511 is displayed in XML format. (Note that the full URL 510 is http://rhacer.libertyodbc.com/Demo/GetItem.asp?FileName=RHQ&ItemId=1; the result is similar to the result of the GetFile.asp, except that the GetItem feature builds a SELECT statement for the filter text).

An XML Abstraction of Enterprise Data Sources

The current invention provides for converting a plurality of legacy data sources to a uniform model of data representation by representing each legacy data source as a virtual in-memory tree-based data structure. The current invention provides a tool that allows the user to graphically explore the data sources in the user's enterprise and to build an XML representation of the Enterprise View of the relevant data sources. The current invention further provides a tool that allows the user to graphically explore the metadata structures and/or content for each particular data source relevant to the Enterprise View and to generate an XML representation of each data source's metadata structure. The tool further builds W3C standards-based syntax for accessing content from each data source, and maps the W3C standards-based access syntax to the access syntax supported by and/or native to each data source.

As an illustrative example, one of the data sources available to a user's enterprise is a database which can be accessed using Microsoft's ActiveX Data Object (ADO). In this illustrative example, the current invention provides a tool that allows the user to graphically explore the ADO database as one of the data sources available to the enterprise. In this illustrative example, using the current invention, the user performs a series of graphical inquiries, such as QBE's (Query by Example, i.e., a point and click technology) to display and select the data sources available to the enterprise, the data bases referenced by each selected data source, the tables referenced by each database, and the rows referenced by each table. For each selected data source, such as the ADO database in our example, the current invention generates an XML representation of the view (the "view element") of that data source and appends it into the enterprise view. For each metadata level selected by the user for the selected data source, the current invention generates as an XML fragment an XML

representation of user selected metadata structures. The current invention appends the XML fragment as a child element of the view element for the data source into the enterprise view. The current invention then generates a W3C XPath syntax necessary to access content in the selected database, which in the illustrative example is the ADO database; the current invention then maps the W3C XPath syntax to American National Standards Institute (ANSI) Structured Query Language (SQL) Syntax.

In one exemplary embodiment of the invention, the current invention provides an application that abstracts a Relational Database Management System (RDBMS) data source as a virtual, in-memory DOM tree.

In the exemplary embodiment, the current invention provides an XML schema Builder. The XML schema builder is a program that prompts the user for a datasource identifier and certain properties, connects the current invention to a specific database catalog, and then provides the user with the ability to apply the datasource identifier, the properties, and the database catalog as part of an XML structure. For example, the XML schema builder prompts the user for an ActiveX Data Object (ADO) datasource and properties. Once the user supplies the requested information, the XML schema builder connects the current invention to a specific Object Linking and Embedding (OLE) Database (DB) Catalog. The current invention then provides for the application of the ADO datasource and properties as well as the OLE DB Catalog to an XML structure.

The current invention prompts the user to supply a Root Element Name for the XML representation. In an illustrative example, the Root Element Name is "myEnterpriseView". As described below, a data model is selected to represent a data source as a DOM tree. In the example described below, an example template for representing data in multiple RDBMS data sources within an enterprise is provided. A person with ordinary skill in the art will understand that the examples provided illustrate an exemplary embodiment, and are not a limitation of the invention.

As previously mentioned above, a Root element can contain attributes. A Root element can also contain an element with which to specify "View" properties about the structure and detailed information about what can be contained in the structure. An illustrative example of a View Root Element is provided below.

```
<myEnterpriseView readWrite="readonly">
    <properties>
        <viewTypes>
            <viewType>ADO</viewType>
```

```
                    <viewType>file:HTML</viewType>
                    <viewType>http:HTML</viewType>
                    <viewType>https:HTML</viewType>
                </viewTypes>
            </properties>
        </myEnterpriseView>
```

The Root Element further contains an element called "views", which contains individual view elements with a name attribute containing the name of the view. As depicted in the illustrative example below, the 'view name" contains a name of a type identifier. The type identifier creates restrictions on the type of datasource that can be viewed.

```
        <myEnterpriseView readWrite="readonly">
            <properties>
                ...
            </properties>
            <views>
              <view name="mySqlServerView" type="ADO" providerString="...">
                    ...
              </view>
            </views>
        </myEnterpriseView>
```

The providerString attribute specified above is the OLE DB Provider String property. The example above demonstrates that in the exemplary embodiment depicted here, various properties can be specified with separate values.

Extending the description of the illustrative example further as depicted below, the XML schema builder provides the user with the ability to reference specific RDBMS elements, for example, catalogs, publishers, tables of authors and titles, and stored procedures.

```
        <myEnterpriseView>
```

```
<properties>
    ...
</properties>
<views>
    <view name="mySqlServerView" type="ADO" providerString="...">
        <catalogs>
        <pubs>
            <tables>
                <authors/>
                <titles/>
                ...
            </tables>
            <storedProcedures>
                ...
            </storedProcedures>
        </pubs>
        </catalogs>
    </view>
</views>
</myEnterpriseView>
```

The current invention provides for View Definition and provides a View Definition Builder. A View Definition is an XML file that describes the <view> elements and their respective immediate attributes. In the exemplary embodiment, the View Definition is static and can be represented in memory.

The View Definition Builder is a program that creates an empty View Definition and then prompts the user to create a new view. The View Definition Builder applies the resulting DOM tree as an ASCII XML file to the Root Element, which in the case of the illustrative

example is, "myEnterpriseView.xml". Additionally, the current invention prompts the user to select the view types, e.g., ADO, file: HTML, etc. The current invention further prompts the user to select a name for the view and to assign all relevant OLE DB properties to establish a connection.

The current invention provides a View Representation Builder. The View Representation Builder is a program that travels through the structure created in the XML schema builder and View Definition Builder functions described above. As the View Representation Builder travels through the View structure, it creates an in-memory DOM tree using the features of the invention previously described above.

To do this, the View Representation Builder uses the view definition item to traverse all views and retrieve all table and column names, and in an alternative embodiment, all stored procedures.

A layered dynamic XML DOM interface is provided. The DOM interface is used as a dynamic layer over other representations of DOM objects. In this way, the current invention creates the effect of an in-memory graph of the metadata and data objects that comprise the enterprise view. This dynamic enterprise view acts in some respects as operating system.

In the preferred embodiment, the current invention provides:

- Support for XPath, XLink, XSL, XSLT, XPointer and other W3C features.
- Full DOM Level 1 support (research level 2)
- Identification of a doc-fault (page-fault) when a reference was made to something that was not in memory; this would trigger dynamic loading.
- Governors that prevent certain users from performing certain combinations of functions. For instance, a restriction prevents a user from applying an XSLT transformation against 3 200-Terabyte Oracle servers and a 3 Terabyte Sql Server database.
- Definition of nested structures (such as, for example, using shape-provider-generated data) as output
- Leverage the MSXML object structure to get the latest in innovations;
- A meta-layer for some of the functionality.

As mentioned above, the current invention provides for layered views. In the exemplary embodiment, the current invention provides for an Enterprise Layer, a Database Metadata Layer, and a Data Record Layer.

The Enterprise Layer is the highest level layer and represents the XML View object.

The XML View object was previously described above.

The Database Metadata Layer represents the metadata describing the databases and tables and table schema of the data source. This layer is dynamic and provides for data loading on demand. The Database Metadata Layer is subordinate to the Enterprise Layer.

The Data Record Layer is subordinate to the Database Metadata Layer. The Data Record Layer represents a row or rows of data from a particular database that is to be searched. When a user receives a result from the server, the result will indicate a rowset, which equates to a nodeList in DOM terms. In one embodiment, at this point, the current invention will not provide the user with the number of nodes (rows) in the nodeList; to get a count, a pass through the entire result set is required. In an alternative embodiment, scrollable cursors on the server are used to obtain a count.

The current invention provides for interaction between the view layers. The current invention provides a mechanism that parses W3C standards-based syntax for accessing XML data and performs the following functions:

- Identify the depth to which traversal is being done
- At whatever depth interaction is happening, prevent recursion into another level without controlling the way in which this is done.
- Apply governors to ensure that too large a set is not invoked.
- Apply a commit mechanism to enable writing the data back.
- Apply a security mechanism that prevents creation of an object at a level to which the user does not have access. For example, if a user applies XPath syntax to the abstracted view of the enterprise, the current invention parses the XPath syntax to determine which portions of it reference the enterprise level definition and which portions of it reference the data source level. The current invention maps the XPath syntax into a Query execution plan that includes steps such as traversing the enterprise level XML, opening connections to data sources, creating data source-specific syntax for accessing data, and building the appropriate output structures.

In the exemplary embodiment, the current invention provides an XML syntax that defines a shape-provided nested table join function. In the exemplary embodiment, this feature uses the ADO Shape Provider.

In an alternative embodiment, the current invention provides a syntax that builds an in-memory tree of data, then performs a commit. The commit invokes a two-phase commit

against the back-end database. In one such embodiment, the current invention requires specification at access time of whether read-only, forward-only, or other specific access logic applies.

Following is an illustrative application example. An application developer is asked to create a system that will allow authors to check sales of their books at a store-by-store basis. The enterprise for which the application developer works has recently gone through a series of acquisitions, which have resulted in the enterprise having two Oracle systems for two of the enterprise's stores, and an SQL Server system. All of these systems are accessible through a Wide Area Network (WAN). In addition, the enterprise has extranet-style access to two additional stores through secure web servers over the Internet.

When an author requests information about the author's book sales, the author wants the application system to provide a single, unified user interface with a single, unified view of the data. An online report that provides the author/user with a table showing two columns, one column identifying each store, and the second column identifying the number of books by that author sold by that store would provide the author/user with the information requested. The online table report would further report a total at the bottom of the second column. Some authors will check sales over the Internet; others will use a wireless device to check sales. In order to facilitate these different devices, the requests and responses are all encoded as XML. In this illustrative example, a combination of Active Server Pages and XSL (XML Stylesheet Language) are used to determine the client type and generate the appropriate WML (Wireless Markup Language)/WAP (Wireless Access Protocol) or HTML/HTTP code for the client device.

Without the present invention, the application developer must first identifies a unified XML layer that will return the appropriate XML regardless of which database or web server that supplies the data. Then, for each back-end, the application developer must create a web-page to extract the appropriate data from the corresponding back end, constructing XML as needed. These back-end access web pages can be used in other projects. Then, the application developer must create the web page that performs the work to load the appropriate results using the intermediate web pages described above to retrieve the data, and to then consolidate the data into a consolidated XML object. This working web page has a single input format and a single XML grammar for the outputs. Finally, the application developer must program the page to identify the type of client and invoke the appropriate XSL style sheet to perform the appropriate conversion for display of the information for the particular client device.

Each back-end has a different access strategy. Oracle and SQL Server are not totally uniform. ADO provides some consistency. However, to obtain maximum performance,

differences in how things are accessed can be a factor. Each server may have different databases, security, tables, and columns. Some of the sources are not even databases, but are web servers. Web page server may provide XML or HTML. The application developers, in addition to knowing the application domain and Visual Basic, must also know ADO, SQL, and XML programming, including using DOM.

The present invention provides a special server extension (portal) that allows application developers to treat databases and web servers as DOM trees. For the example author information request illustrated above, the following illustrative representation is provided:

```
<myEnterprise>
    <views>
        <view name="SqlServerView" type="ADO" ...>
            <catalogs>
                <pubs>
                    <titles>
                        ...
                    </titles>
                </pubs>
            </catalogs>
        </view>
        <view name="OracleABC" type="ADO" ...>
            <catalogs>
                <pubs>
                    <titles>
                        ...
                    </titles>
                </pubs>
            </catalogs>
```

```
</view>
<view name="OracleABC" type="ADO" ...>
    <catalogs>
        <pubs>
            <titles>
                ...
            </titles>
        </pubs>
    </catalogs>
</view>
<view name="Web1" type="HTML" ...>
    <pages>
        <getTitles.jhtml>
            <titles>
                ...
            </titles>
        </getTitles.jhtml>
    </pages>
</view>
<view name="Web2" type="XML" ...>
    <pages>
        <getTitles.asp>
            <titles>
                ...
            </titles>
```

```
                    </getTitles.asp>
                </pages>
            </view>
        </views>
    </myEnterprise>
```

In this example, for titles, there is always an auth_id field element. Using the node path syntax previously described above, the following example methods would retrieve all titles rows for which the auth_id attribute has a value of "123":

```
1.)  set objBizCollection=
            objBizDOMPlus.query("//titles/row[@auth_id='123']")
     or,
2.)  For Each objBizNode in
objBizDOMPlus.xquery("//titles/row[@auth_id='123']")
            ' do something with it
     Next
```

If the target sources do not share the same field names, the method would need to use an "or" predicate to find the appropriate values. An SQL Query or filter is applied to return a result if it is determined from metadata that the appropriate fields exist to support the query.

A fully qualified XPath is used in some embodiments to retrieve the results from a specific data source so that other data sources do not have to be opened and checked. In one embodiment, the current invention instantiates a connection, only as needed, and utilizes OLE DB resource pooling, where available, to achieve maximum performance and scalability.

Business application programmer/developers trying to integrate line-of-business Legacy systems with net market transactions are faced with barriers including varied data access programming models, different data structures, and varied data types. In the exemplary embodiment of the invention, a data integration server is programmed to provide access to Line-Of-Business Legacy data from such sources as RDBMS, HTTP, FTP, COM, CORBA and others; the server is programmed to use Internet standards. The exemplary embodiment provides eBusiness application developers with the ability to leverage W3C Industry standards to integrate varied data sources by providing a uniform model of data

representation, a uniform data access programming model, and uniform datatyping.

Relational Database Management systems (RDBMS) have a common logical structure, which has been powerfully abstracted by existing database middleware products. FIG. 46 depicts an exemplary tree representation of an exemplary RDBMS configuration.

The current invention can instantiate a tree from any single node in the larger tree. For purposes of the following description of the invention, "virtualized" refers to a portion of a DOM tree that is supplied by a data source. "instantiate" means the act of creating the relevant portion of the DOM tree in memory. "serialize" means the act of writing virtualized portions of data back to the data source.

The current invention provides sub-tree instantiation by providing the user with the option of deciding at which point in the tree, the tree is to be instantiated.

The current invention further provides a virtual in-memory tree and virtual instantiation. The user asks for a property of a current node. The current invention checks the status of the current node and reports the status as: 1.) it is from a static layer currently in memory; or 2.) it represents a layer that is supplied by a data source and is a.) currently in memory; or b.) it is not in memory at which point the current invention reads the node and loads it into memory.

The current invention manages node write-back. When a programmer performs method calls that create nodes in areas of the Virtual DOM that are virtualized, the current invention instantiates the virtualized portion of the Virtual DOM. When a virtualized portion of the virtualized DOM is instantiated, the current invention sets flags to indicate for each node, whether the data has been changed. Each flag has four possible states: "0" represents No Change; "1" represents New; "2" represents Changed; and "3" represents "Deleted". The values 0-4 are illustrative and not a limitation of the invention.

The current invention provides a serialize method and a discardChanges method. The serialize method starts a transaction that serializes changes to the data source and commits the transaction. When the serialize method is used, only the relevant nodes are written out. In the exemplary embodiment, references to writeable nodes are kept in a hash table for high-speed access at write time.

The discardChanges method returns the in-memory tree to its pre-change status. To support the discardChanges method, when a node is first changed, the current invention saves the original values in an "invisible" subtree. When a discardChanges method is issued, the current invention retrieves the original values from the "invisible" subtree and replaces the original values in the node.

When a node is deleted, the current invention sets the node's status to "invisible".

When a new node is requested, the current invention creates a new node.

In an alternative embodiment, the current invention starts a transaction and then allows all changes to write to the data source. In this embodiment, when the serialize method is used, the current invention then commits the transaction. In this embodiment, the discardChanges method simply rolls back the transaction.

ILLUSTRATIVE EMBODIMENTS

Although this invention has been described in certain specific embodiments, many additional modifications and variations would be apparent to those skilled in the art. It is, therefore, to be understood that this invention may be practiced otherwise than as specifically described. Thus, the embodiments of the invention described herein should be considered in all respects as illustrative and not restrictive, the scope of the invention to be determined by the appended claims and their equivalents rather than the foregoing description.

WHAT IS CLAIMED IS:

1. A method using a computer for converting source data to an Internet data structure, wherein said source data is characterized by a particular structure and by a set of relationships, wherein the structure and the set of relationships are described by metadata, and wherein said metadata comprises a plurality of metadata components, and wherein said metadata and said metadata components are accessible by said computer, and wherein said Internet data structure is characterizable by a tree-based structure definition, the method comprising:

translating the metadata into a set of DTD declarations and a corresponding tree-based structure definition, wherein said tree-based structure definition comprises a plurality of node definitions and wherein each DTD declaration is equivalent to one of said metadata components and wherein each of said node definitions is equivalent to one of said metadata components;

storing as one of a plurality of conversion map components each equivalent document type definition declaration and the corresponding metadata component;

storing as one of a plurality of conversion map components each equivalent node definition and the corresponding metadata component;

aggregating the conversion map components into a conversion map; and

mapping the source data into the Internet data structure according to the conversion map.

2. A computer system programmed for converting source data to an Internet data structure, wherein said source data is characterized by a particular structure and by a set of relationships, wherein the structure and the set of relationships are described by metadata, and wherein said metadata comprises a plurality of metadata components, and wherein said metadata and said metadata components are accessible by said computer, and wherein said Internet data structure is characterizable by a tree-based structure definition, the computer system comprising:

program code for translating the metadata into a set of document type definition declarations and a corresponding tree-based structure definition, wherein said tree-based structure definition comprises a plurality of node definitions and wherein each document type definition declaration is equivalent to one of said metadata components and wherein each of said node definitions is equivalent to one of said metadata components;

program code for storing as one of a plurality of conversion map components each equivalent document type definition declaration and the corresponding metadata component;

program code for storing as one of a plurality of conversion map components each equivalent node definition and the corresponding metadata component;

program code for aggregating the conversion map components into a conversion map; and

program code for mapping the source data into the Internet data structure according to the conversion map.

3. A method using a computer of converting source legacy data to an Internet data structure, the method comprising:

converting the source legacy data to a marked up electronic document; and

using a marked up electronic document parser program to transform the marked up electronic document to a machine readable tree-based format.

4. The method of claim 3 wherein the marked up electronic document is an XML document.

5. The method of claim 4 wherein the machine readable tree-based format is a Document Object Model.

6. A method using a computer of converting source legacy data to an Internet data structure, the method comprising:

determining a plurality of relationships between the source legacy data and the Internet data structure; and

converting the source legacy data to a marked up electronic document according to the determined relationships.

7. The method of claim 6 wherein the marked up electronic document is marked up with tags, elements and attributes that reflect the data and the intent of the content of the

source legacy data.

8. The method of claim 7, the method further comprising:

accepting as input the marked up electronic document; and

producing as output the content of the marked up electronic document in machine readable tree-based form.

9. A method using a computer of converting source legacy data to an Internet data structure, the method comprising:

converting the legacy data to an XML document; and

using an XML parser to transform the XML document to a machine readable tree-based DOM.

10. A method using a computer of converting source Internet data to a legacy data structure, the method comprising:

determining a plurality of relationships between the source Internet data and the legacy data structure; and

converting the source Internet data to the legacy data structure.

11. The method of claim 10 wherein the converted Internet data in the legacy data structure reflects all of the data and the intent of the content of the source Internet data.

12. A method using a computer of integrating an Internet data source with legacy data structures, the method comprising:

converting the Internet data source to a simple legacy data structure wherein said source Internet data is in the form of a machine readable tree-based document object; and

integrating the converted Internet data into the legacy data structure.

13. The method of claim 12 wherein said Internet data source is in the form of a machine readable tree-based Document Object Model.

14. The method of claim 13 wherein said Document Object Model has a plurality of nodes, said nodes arranged according to a tree-based hierarchy, each level of said hierarchy having associated with it a unique delimiter.

15. A method using a computer of integrating a first source of Internet data having a first data structure wherein said first data structure having a plurality of data fields and wherein each data field of said first data structure having a tag name, with a second source of Internet data having a second data structure wherein said second data structure having a plurality of data fields and wherein each field having a tag name, the method comprising:

identifying each data element in said first data structure that corresponds to at least one data field in said second data structure; and

renaming each data element in said first data structure with the name of the corresponding data field in said second data structure.

16. The method of claim 15 further comprising:

building a record comprising data from said renamed first data structure; and

labeling each data element in the record with the corresponding name of the corresponding data field in the second data structure.

17. The method of claim 16 further comprising the step of:

posting the record to said second data structure.

18. The method of claim 15 wherein the renaming step is accomplished using a Windows Scripting Hosted language.

19. The method of claim 18 wherein the Windows Scripting Hosted language is

Visual Basic Script.

20. The method of claim 18 wherein the Windows Scripting Hosted language is JavaScript.

21. The method of claim 18 wherein the Windows Scripting Hosted language is PerlScript.

22. A method using a computer of converting source Internet data to a simple legacy data structure wherein said source Internet data is in the form of a machine readable tree-based Document Object Model with a plurality of nodes said nodes arranged according to a tree-based hierarchy, each level of said hierarchy having associated with it a unique delimiter, the method comprising:

identifying a root node of the Document Object Model;

traversing each child level node immediately subordinate to the root node;

traversing every node subordinate to each child node before proceeding to the next child node;

inserting all of the data fields contained in each node subordinate to each child node into the simple legacy data structure; and

inserting into the simple legacy data structure a delimiter for each data field.

23. A method using a computer of converting source legacy data to an Internet data structure, wherein said source legacy data is comprised of one or more records of data, each record comprising a first hierarchical level of data wherein said first hierarchical level of data comprises at least one subordinate hierarchical level of data, wherein each hierarchical level of the source legacy data contains at least one data field, wherein a plurality of successively hierarchical levels in the Internet data structure have been defined and wherein for each of said Internet data hierarchical levels, a unique start-of-data name tag and a unique end-of-data name tag have been assigned, for each source legacy data record the method comprising:

initializing an output conversion record;

inserting in the conversion record the unique start-of-data name tag for the first Internet data hierarchical level;

identifying a next successive hierarchical level within the said source legacy data record;

identifying a hierarchical level from among the Internet data hierarchical levels that is equivalent in hierarchy level to the source legacy data hierarchical level;

inserting in the conversion record the unique start-of-data name tag of the corresponding Internet data hierarchical level;

identifying each data field contained in the source legacy data hierarchical level;

transferring all of the data fields contained in the source legacy data hierarchical level to the conversion record immediately following the unique start of data name tag of the corresponding Internet data hierarchical level;

inserting in the conversion record the unique end of data name tag of the corresponding Internet data hierarchical level; and

repeating the steps until all of the data contained in the legacy data record has been converted.

24. A method of converting source data structured according to a simple data structure to a tree-based rich data structure, wherein said source data is comprised of one or more records of data, each record comprising a first hierarchical level of data wherein said first hierarchical level of data comprises at least one subordinate hierarchical level of data, the method comprising:

identifying each hierarchical level within said source data;

for each of said source data hierarchical levels, creating an element level node in said target data structure corresponding to said source data hierarchical level;

for each of said source data hierarchical levels, identifying all items of source data wherein said source data is directly associated with said source data hierarchical level; and

for each of said source data hierarchical levels, saving in said target data structure all of said identified items of data directly associated with said source data hierarchical level and designating said saved data as being associated with said corresponding target data element level node.

25. A method using a computer of converting source data structured according to a simple data structure to target data stored structured according to a tree-based data structure having at least one relational node, the method comprising:

creating a first node in said target data store;

identifying at least one key level within said source data;

for each of said source key levels, creating a node in said target data corresponding to said source key level;

associating each created node in said target data with at least one other node in said target data store;

for each of said source key levels, identifying all items of data wherein said data is directly associated with said source key level;

for each of said source key levels, describing all items of data directly associated with said source key level in terms of said target data node; and

for each of said source key levels, saving in said target data store all said described items of data directly associated with said source key level in terms of said target data node.

26. A computer system for converting source data to an Internet data structure, wherein said source data is characterized by a particular structure and by a set of relationships, wherein the structure and the set of relationships are described by metadata, and wherein said metadata comprises a plurality of metadata components, and wherein said metadata and said metadata components are accessible by said computer, and wherein said Internet data structure is characterizable by a tree-based structure definition, the computer system programmed to:

translate the metadata into a set of document type definition declarations and a corresponding tree-based structure definition, wherein said tree-based structure definition comprises a plurality of node definitions and wherein each document type definition declaration is equivalent to one of said metadata components and wherein each of said node definitions is equivalent to one of said metadata components;

store as one of a plurality of conversion map components each equivalent document type definition declaration and the corresponding metadata component;

store as one of a plurality of conversion map components each equivalent node definition and the corresponding metadata component;

aggregate the conversion map components into a conversion map; and

convert the source data into the Internet data structure according to the conversion map.

27. A computer program product having instructions for:

converting without data loss a legacy data source to a tree-based Internet data structure; and

integrating without data loss said converted legacy data with an existing Internet data source, said existing Internet data source having a tree-based Internet data structure .

28. A computer program product having instructions for:

converting without data loss an Internet data source having a tree-based Internet data structure to a simple data structure; and

integrating without data loss said converted Internet data with an existing legacy data source, said legacy data source having a simple data structure.

29. A computer program product having instructions for:

exposing the ability to execute a method on a Legacy Line-of-Business database server using an Internet application on a Web server; and

returning the results of the executed method to the Web server as an Internet data structure.

30. A computer program product having instructions for:

wrapping a first Document Object Model with a second Document Object Model.

31. The computer program product of Claim 30 having further instructions for:

exposing to the second Document Object Model a plurality of properties and a plurality of methods which define a plurality of aspects of said first Document Object Model.

32. A computer program product having instructions for:

creating a first Document Object Model with a first class definition as a template for a second Document Object Model; and

creating the second Document Object Model with a second class definition.

33. The computer program product of Claim 32 having further instructions for:

replacing the second class definition of the second Document Object Model with the first class definition of the first Document Object Model.

34. The computer program product of Claim 33 having further instructions for:

loading the first Document Object Model into the second Document Object Model.

35. The computer program product of Claim 34 having further instructions for:

exposing to the second Document Object Model a plurality of properties and a plurality of methods which define a plurality of aspects of said first Document Object Model.

36. The computer program product of Claim 35 having further instructions for:

exposing to the second Document Object Model a plurality of data items contained within said first Document Object Model.

37. The computer program product of Claim 35 having further instructions for:

operating on the second Document Object Model to determine a value for at least one of said exposed properties.

38. The computer program product of Claim 35 having further instructions for:

operating on the second Document Object Model to establish a value for at least one of said exposed properties.

39. The computer program product of Claim 35 having further instructions for:

operating on a method that controls an aspect of the second Document Object Model to control an aspect of the first Document Object Model.

40. The computer program product of Claim 36 having further instructions for:

operating on the second Document Object Model to add a data item to said first Document Object Model.

41. The computer program product of Claim 36 having further instructions for:

operating on the second Document Object Model to remove a data item from the data contained within said first Document Object Model.

42. The computer program product of Claim 36 having further instructions for:

operating on the second Document Object Model to modify a data item contained within said first Document Object Model.

43. The computer program product of Claim 36 having further instructions for:

operating on the second Document Object Model to retrieve a data item contained within said first Document Object Model.

44. The computer program product of Claim 36 having further instructions for:

exposing to the second Document Object Model a plurality of structural elements contained within said first Document Object Model.

45. The computer program product of Claim 44 having further instructions for:

operating on the second Document Object Model to add a structural element to said first Document Object Model.

46. The computer program product of Claim 44 having further instructions for:

operating on the second Document Object Model to remove a structural element from the structural elements contained within said first Document Object Model.

47. The computer program product of Claim 44 having further instructions for:

operating on the second Document Object Model to modify a structural element contained within said first Document Object Model.

48. The computer program product of Claim 44 having further instructions for:

operating on the second Document Object Model to retrieve a structural element contained within said first Document Object Model.

49. The computer program product of Claim 44 having further instructions for:

operating on the second Document Object Model to retrieve a collection of a plurality of structural elements contained within said first Document Object Model.

50. The computer program product of Claim 44 having further instructions for:

specifying a name to the second Document Object Model;

operating on the second Document Object Model to identify a component of said first Document Object Model as a particular property, method, data item or structural component.

51. A computer program product for providing a browser Q-Pointer interface having instructions for:

receiving input from a user containing new Q-Pointer information for an existing file; and

tying said user input Q-Pointer information to said file.

52. A computer program product for providing a browser Q-Pointer interface

having instructions for:

reporting to a user a plurality of Q-Pointer information for at least one existing file.

53. The computer program product of Claim 52 having further instructions for:

receiving input from a user for modifying said Q-Pointer information.

54. A computer program product for accessing a set of hierarchical data using a global communications network server programming environment, said computer program product having instructions for:

establishing a browser interface connection;

retrieving a file name variable;

retrieving a filter text value; and

retrieving said set of hierarchical data using said file name variable and said filter text value.

55. A computer system programmed for accessing and integrating electronic data, said computer system programmed to:

wrap a first Document Object Model with a second Document Object Model.

56. The computer system of Claim 55 further programmed to:

expose to the second Document Object Model a plurality of properties and a plurality of methods which define a plurality of aspects of said first Document Object Model.

57. A computer system programmed for accessing and integrating electronic data, said computer system programmed to:

create a first Document Object Model with a first class definition as a template for a second Document Object Model; and

create the second Document Object Model with a second class definition.

58. The computer system of Claim 57 further programmed to:

replace the second class definition of the second Document Object Model with the first class definition of the first Document Object Model.

59. The computer system of Claim 58 further programmed to:

load the first Document Object Model into the second Document Object Model.

60. The computer system of Claim 59 further programmed to:

expose to the second Document Object Model a plurality of properties and a plurality of methods which define a plurality of aspects of said first Document Object Model.

61. The computer system of Claim 60 further programmed to:

expose to the second Document Object Model a plurality of data items contained within said first Document Object Model.

62. The computer system of Claim 60 further programmed to:

operate on the second Document Object Model to determine a value for at least one of said exposed properties.

63. The computer system of Claim 60 further programmed to:

operate on the second Document Object Model to establish a value for at least one of said exposed properties.

64. The computer system of Claim 60 further programmed to:

operate on a method that controls an aspect of the second Document Object Model to

control an aspect of the first Document Object Model.

65. The computer system of Claim 60 further programmed to:

operate on the second Document Object Model to add a data item to said first Document Object Model.

66. The computer system of Claim 61 further programmed to:

operate on the second Document Object Model to remove a data item from the data contained within said first Document Object Model.

67. The computer system of Claim 61 further programmed to:

operate on the second Document Object Model to modify a data item contained within said first Document Object Model.

68. The computer system of Claim 61 further programmed to:

operate on the second Document Object Model to retrieve a data item contained within said first Document Object Model.

69. The computer system of Claim 61 further programmed to:

expose to the second Document Object Model a plurality of structural elements contained within said first Document Object Model.

70. The computer system of Claim 69 further programmed to:

operate on the second Document Object Model to add a structural element to said first Document Object Model.

71. The computer system of Claim 69 further programmed to:

operate on the second Document Object Model to remove a structural element from the structural elements contained within said first Document Object Model.

72. The computer system of Claim 69 further programmed to:

operate on the second Document Object Model to modify a structural element contained within said first Document Object Model.

73. The computer system of Claim 69 further programmed to:

operate on the second Document Object Model to retrieve a structural element contained within said first Document Object Model.

74. The computer system of Claim 69 further programmed to:

operate on the second Document Object Model to retrieve a collection of a plurality of structural elements contained within said first Document Object Model.

75. The computer system of Claim 69 further programmed to:

specify a name to the second Document Object Model;

operate on the second Document Object Model to identify a component of said first Document Object Model as a particular property, method, data item or structural component.

76. A computer system for providing a browser Q-Pointer interface, said computer system programmed to:

receive input from a user containing new Q-Pointer information for an existing file; and

tie said user input Q-Pointer information to said file.

77. A computer system for providing a browser Q-Pointer interface, said computer system programmed to:

report to a user a plurality of Q-Pointer information for at least one existing file.

78. The computer system of Claim 77 further programmed to:

receive input from a user for modifying said Q-Pointer information.

79. A computer system for accessing a set of hierarchical data using a global communications network server programming environment, said computer system programmed to:

establish a browser interface connection;

retrieve a file name variable;

retrieve a filter text value; and

retrieve said set of hierarchical data using said file name variable and said filter text value.

80. A method using a computer for accessing and integrating electronic data, said method comprising:

creating a first Document Object Model with a first class definition as a template for a second Document Object Model; and

creating the second Document Object Model with a second class definition.

81. The method of Claim 80 further comprising:

replacing the second class definition of the second Document Object Model with the first class definition of the first Document Object Model.

82. The method of Claim 81 further comprising:

loading the first Document Object Model into the second Document Object Model.

83. The method of Claim 82 further comprising:

exposing to the second Document Object Model a plurality of properties and a plurality of methods which define a plurality of aspects of said first Document Object Model.

84. The method of Claim 83 further comprising:

exposing to the second Document Object Model a plurality of data items contained within said first Document Object Model.

85. A method using a computer for providing a browser Q-Pointer interface, said method comprising:

receiving input from a user containing new Q-Pointer information for an existing file; and

tying said user input Q-Pointer information to said file.

86. A method using a computer for providing a browser Q-Pointer interface, said method comprising:

reporting to a user a plurality of Q-Pointer information for at least one existing file.

87. The method of Claim 86 further comprising:

receiving input from a user for modifying said Q-Pointer information.

88. A method using a computer for accessing a set of hierarchical data using a global communications network server programming environment, said method comprising:

establishing a browser interface connection;

retrieving a file name variable;

retrieving a filter text value; and

retrieving said set of hierarchical data using said file name variable and said filter text value.

89. A computer system for accessing and integrating electronic data, said computer system comprising:

a set of program instructions for creating a first Document Object Model with a first class definition as a template for a second Document Object Model; and

a set of program instructions for creating the second Document Object Model with a second class definition.

90. The computer system of Claim 89 further comprising:

a set of program instructions for replacing the second class definition of the second Document Object Model with the first class definition of the first Document Object Model.

91. The computer system of Claim 90 further comprising:

a set of program instructions for loading the first Document Object Model into the second Document Object Model.

92. The computer system of Claim 91 further comprising:

a set of program instructions for exposing to the second Document Object Model a plurality of properties and a plurality of methods which define a plurality of aspects of said first Document Object Model.

93. The computer system of Claim 92 further comprising:

a set of program instructions for exposing to the second Document Object Model a plurality of data items contained within said first Document Object Model.

94. A computer system for providing a browser Q-Pointer interface, said computer system comprising:

a set of program instructions for receiving input from a user containing new Q-Pointer information for an existing file; and

a set of program instructions for tying said user input Q-Pointer information to said

file.

95. A computer system for providing a browser Q-Pointer interface, said computer system comprising:

a set of program instructions for reporting to a user a plurality of Q-Pointer information for at least one existing file.

96. The computer system of Claim 95 further comprising:

a set of program instructions for receiving input from a user for modifying said Q-Pointer information.

97. A computer system for accessing a set of hierarchical data using a global communications network server programming environment, said computer system comprising:

a set of program instructions for establishing a browser interface connection;

a set of program instructions for retrieving a file name variable;

a set of program instructions for retrieving a filter text value; and

a set of program instructions for retrieving said set of hierarchical data using said file name variable and said filter text value.

98. A computer program product for providing a Document Object Model language syntax with which to link multiple programming activities in a single line of coding, said computer program product having instructions for:

performing a first program instruction on an object;

returning as a value the object upon which the program instruction was performed;

making said object value available to a second program instruction.

99. A method using a computer of converting a plurality of legacy data sources to

a uniform model of data representation, the method comprising:

representing each legacy data source as a virtual in-memory tree-based data structure.

100. A computer system for converting a plurality of legacy data sources to a uniform model of data representation, the computer system comprising:

a set of program instructions for representing each legacy data source as a virtual in-memory tree-based data structure.

101. A computer system for converting a plurality of legacy data sources to a uniform model of data representation, said computer system programmed to:

represent each legacy data source as a virtual in-memory tree-based data structure.

102. A computer program product for converting a plurality of legacy data sources to a uniform model of data representation, the computer program product having instructions for:

representing each legacy data source as a virtual in-memory tree-based data structure.

103. A method for processing a document, comprising:

providing at least one process document including processing instructions;

accepting an incoming document;

reading a server document, the server document including process document selection instructions for selecting a process document based on the incoming document;

selecting a process document using the server document process document selection instructions and the incoming document; and

processing the incoming document according to the processing instructions in the selected process document.

104. The method of Claim 103 wherein:

the server document is a XML document; and

the selected process document is a XML document.

105. The method of Claim 103, wherein the selected process document processing instructions further include incoming document to working document translation instructions.

106. The method of Claim 105, wherein the selected process document processing instructions further include:

working document to transmission document translation instructions; and

transmitting instructions for transmitting the transmission document via a communication network.

107. The method of Claim 103, further comprising transforming the incoming document into an incoming software object.

108. The method of Claim 107, wherein the selected process document processing instructions further include invoking a document processing software object.

109. A method for processing a document received from a client via a communication network, comprising:

providing at least one process document including processing instructions;

providing at least one translation document;

receiving an incoming document from the client via the communication network;

reading a server document, the server document including:

process document selection instructions for selecting a process document based on the incoming document; and

translation document selection instructions for selecting a translation document based on the incoming document;

selecting a translation document using the server document translation document selection instructions and the incoming document;

translating the incoming document into a working document using the selected translation document;

selecting a process document using the server document process document selection instructions and the incoming document; and

processing the working document according to the processing instructions in the selected process document.

110. The method of Claim 109, wherein the selected process document processing instructions further include transmitting instructions for transmitting the working document via the communication network.

111. The method of Claim 109, wherein the selected process document processing instructions further include:

working document to transmission document translation instructions; and

transmitting instructions for transmitting the transmission document via the communication network.

112. The method of Claim 109, further comprising transforming the working document into a working software object.

113. The method of Claim 109, further comprising transforming the incoming document into an incoming software object.

114. The method of Claim 109, wherein the selected process document processing instructions further include invoking a document processing software object.

115. The method of Claim 109 wherein:

the serving document is an XML document; and

the selected process document is a XML document.

116. The method of Claim 109 wherein:

the incoming document is an XML document;

the working document is an XML document; and

the selected translation document is a XSLT document.

117. A data processing system adapted to process documents received from a client via a communication network, comprising:

a processor; and

a memory operably coupled to the processor and having program instructions stored therein, the processor being operable to execute the program instructions, the program instructions including:

pipeline instructions, the pipeline instructions including:

receiving an incoming document sent by the client via the communications network;

invoking a sorter, the sorter containing instructions, the instructions including selecting processing instructions for the incoming document; and

invoking a dispatcher, the dispatcher containing instructions, the instructions including processing the incoming document according to the selected processing instructions.

118. The data processing system of Claim 117, the processing instructions further including transmitting a document to the client via the communication network.

119. The data processing system of Claim 118, the processing instructions further including transmitting a document to a server via the communication network.

120. A data processing system adapted to process a document, comprising:

a processor; and

a memory operably coupled to the processor and having program instructions stored therein, the processor being operable to execute the program instructions, the program instructions including:

accepting an incoming document;

reading a server document, the server document including process document selection instructions for selecting a process document based on the incoming document, the process document including processing instructions;

selecting a process document using the server document process document selection instructions and the incoming document; and

processing the incoming document according to the processing instructions in the selected process document.

121. The data processing system of Claim 120 wherein:

the server document is a XML document; and

the selected process document is a XML document.

122. The data processing system of Claim 120, wherein the selected process document processing instructions further include incoming document to working document translation instructions.

123. The data processing system of Claim 120, wherein the selected process document processing instructions further include:

working document to transmission document translation instructions; and

transmitting instructions for transmitting the transmission document via a communication network.

124. The data processing system of Claim 120, the program instructions further including transforming the incoming document into an incoming software object.

125. The data processing system of Claim 120, wherein the selected process document processing instructions further include invoking a document processing software object.

126. A data processing system adapted to process a document sent from a client via a communication network, comprising:

a processor; and

a memory operably coupled to the processor and having program instructions stored therein, the processor being operable to execute the program instructions, the program instructions including:

receiving an incoming document sent by the client via the communication network;

reading a server document, the server document including:

process document selection instructions for selecting a process document based on the incoming document, the process document including processing instructions; and

translation document selection instructions for selecting a translation document based on the incoming document;

selecting a translation document using the server document translation document selection instructions and the incoming document;

translating the incoming document into a working document using the selected translation document;

selecting a process document using the server document process document selection instructions and the incoming document; and

processing the working document according to the processing instructions in the selected process document.

127. The data processing system of Claim 126, wherein the selected process document processing instructions further include transmitting instructions for transmitting the working document via the communication network.

128. The data processing system of Claim 126, wherein the selected process document processing instructions further include:

working document to transmission document translation instructions; and

transmitting instructions for transmitting the transmission document via the communication network.

129. The data processing system of Claim 126, the program instructions further comprising transforming the working document into a working software object.

130. The data processing system of Claim 126, the program instructions further comprising transforming the incoming document into an incoming software object.

131. The data processing system of Claim 126, wherein the selected process document processing instructions further include invoking a document processing software object.

132. The data processing system of Claim 126 wherein:

the serving document is a XML document; and

the selected process document is a XML document.

133. The data processing system of Claim 126 wherein:

the incoming document is a XML document;

the working document is a XML document; and

the selected translation document is a XSLT document.

134. A computer program product embodying computer program instructions for execution by a computer, the computer program instructions comprising:

pipeline instructions, the pipeline instructions including:

receiving an incoming document sent by the client via the communications network;

invoking a sorter, the sorter containing instructions including selecting processing instructions for the incoming document; and

invoking a dispatcher, the dispatcher containing instructions including processing the incoming document according to the selected processing instructions.

135. The computer program product of Claim 134, the processing instructions further including transmitting a document to the client via the communication network.

136. The computer program product of Claim 134, the processing instructions further including transmitting a document to a server via the communication network.

137. A computer program product embodying computer program instructions for execution by a computer, the computer program instructions comprising:

accepting an incoming document;

reading a server document, the server document including process document selection instructions for selecting a process document based on the incoming document, the process document containing processing instructions;

selecting a process document using the server document process document selection instructions and the incoming document; and

processing the incoming document according to the processing instructions in the selected process document.

138. The computer program product of Claim 137 wherein:

the server document is a XML document; and

the selected process document is a XML document.

139. The computer program product of Claim 137, wherein the selected process document processing instructions further include incoming document to working document translation instructions.

140. The computer program product of Claim 137, wherein the selected process document processing instructions further include:

working document to transmission document translation instructions; and

transmitting instructions for transmitting the transmission document via a communication network.

141. The computer program product of Claim 137, the computer program instructions further comprising transforming the incoming document into an incoming software object.

142. The computer program product of Claim 137, wherein the selected process document processing instructions further include invoking a document processing software object.

143. A computer program product embodying computer program instructions for execution by a computer, the computer program instructions comprising:

receiving an incoming document sent by the client via the communication network;

reading a server document, the server document including:

process document selection instructions for selecting a process document based on the incoming document, the process document including processing instructions; and

translation document selection instructions for selecting a translation document based on the incoming document;

selecting a translation document using the server document translation document selection instructions and the incoming document;

translating the incoming document into a working document using the selected translation document;

selecting a process document using the server document process document selection instructions and the incoming document; and

processing the working document according to the processing instructions in the selected process document.

144. The computer program product of Claim 143, wherein the selected process document processing instructions further include transmitting instructions for transmitting the working document via the communication network.

145. The computer program product of Claim 143, wherein the selected process document processing instructions further include:

working document to transmission document translation instructions; and

transmitting instructions for transmitting the transmission document via the communication network.

146. The computer program product of Claim 143, the computer program instructions further comprising transforming the working document into a working software object.

147. The data processing system of Claim 143, the computer program instructions further comprising transforming the incoming document into an incoming software object.

148. The computer program product of Claim 143, wherein the selected process document processing instructions further include invoking a document processing software object.

149. The computer program product of Claim 143 wherein:

the serving document is a XML document; and

the selected process document is a XML document.

150. The computer program product of Claim 143 wherein:

the incoming document is a XML document;

the working document is a XML document; and

the selected translation document is a XSLT document.

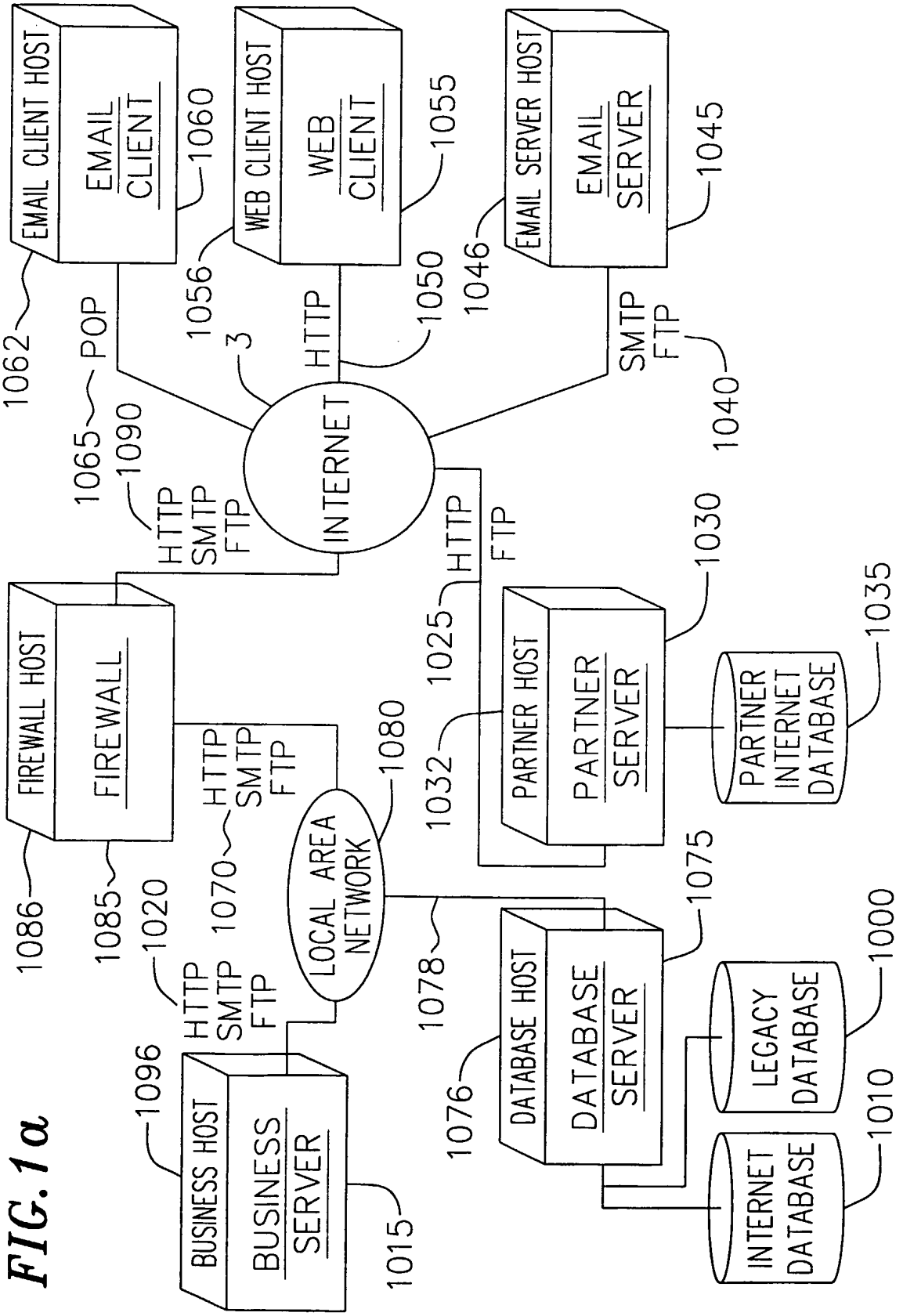
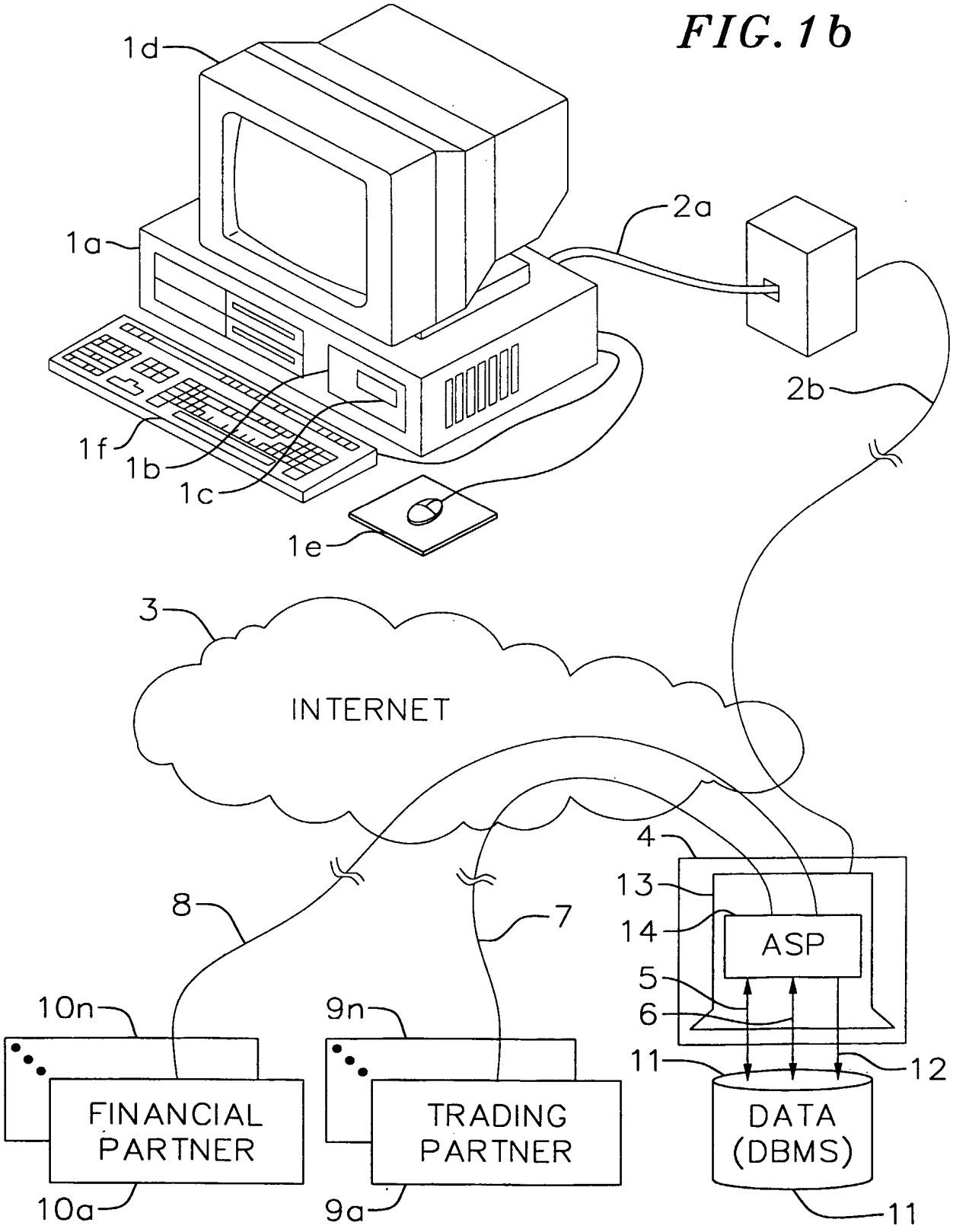

EMAIL CLIENT HOST
EMAIL CLIENT
1060
1062
POP
1065
1090
HTTP
SMTP
FTP
WEB CLIENT HOST
WEB CLIENT
1055
1056
3
HTTP
1050
EMAIL SERVER HOST
EMAIL SERVER
1045
1046
SMTP
FTP
1040
INTERNET
FIREWALL HOST
FIREWALL
1086
1085
1020
HTTP
SMTP
FTP
1070
LOCAL AREA NETWORK
1080
1078
HTTP
FTP
1025
PARTNER HOST
PARTNER SERVER
1030
1032
PARTNER INTERNET DATABASE
1035
BUSINESS HOST
BUSINESS SERVER
1096
1015
DATABASE HOST
DATABASE SERVER
1076
1075
LEGACY DATABASE
1000
INTERNET DATABASE
1010

FIG. 1a

FIG. 1b


1d
1a
2a
2b
1f
1b
1c
1e
3
INTERNET
4
13
14
ASP
5
6
8
7
10n
9n
11
12
FINANCIAL PARTNER
TRADING PARTNER
DATA (DBMS)
10a
9a
11

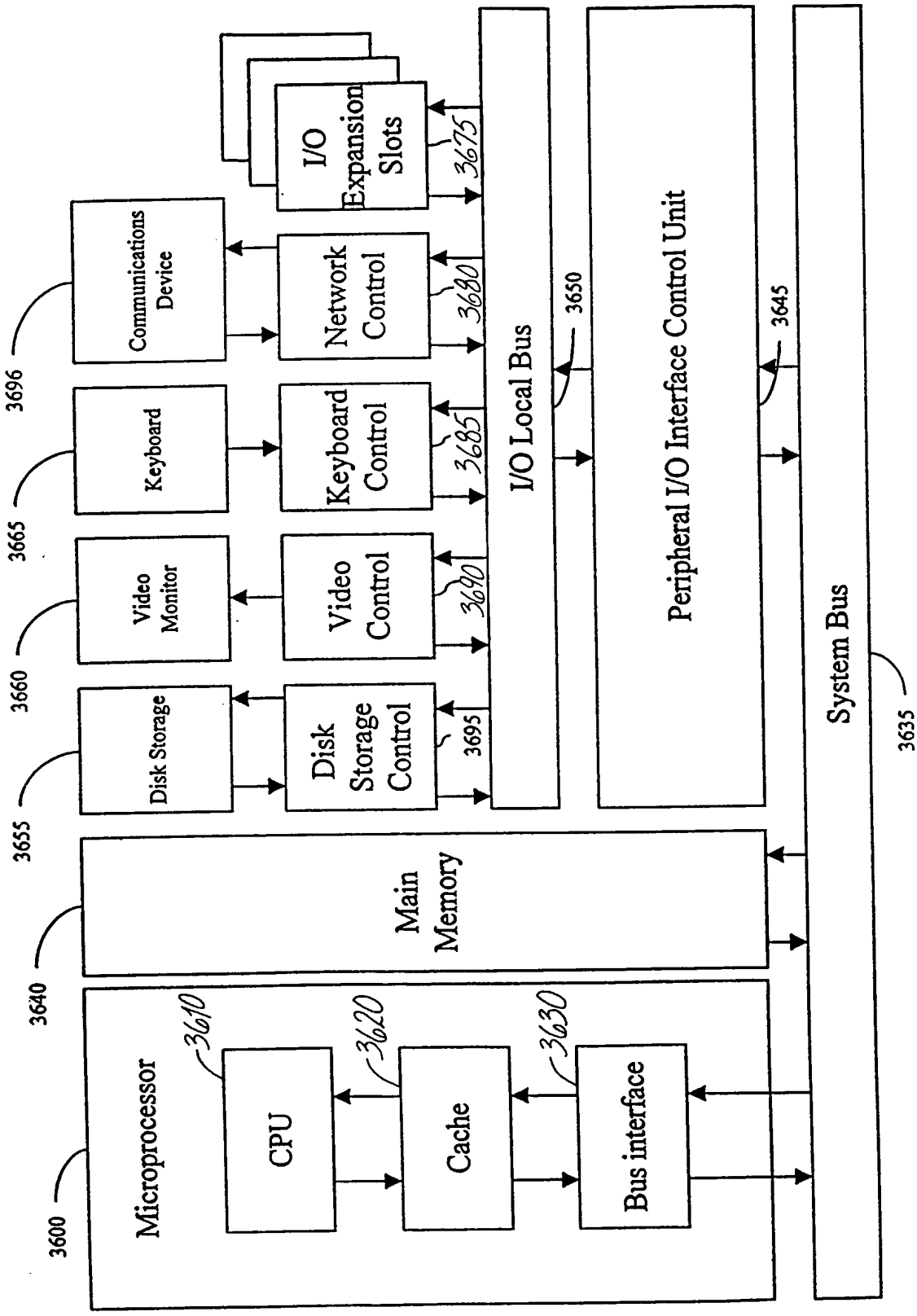

3600
Microprocessor
3610
CPU
3620
Cache
3630
Bus interface
3640
Main Memory
3655
Disk Storage
3660
Video Monitor
3665
Keyboard
3696
Communications Device
Disk Storage Control
3695
Video Control
3690
Keyboard Control
3685
Network Control
3680
I/O Expansion Slots
3675
I/O Local Bus
3650
Peripheral I/O Interface Control Unit
3645
System Bus
3635

FIG. 2

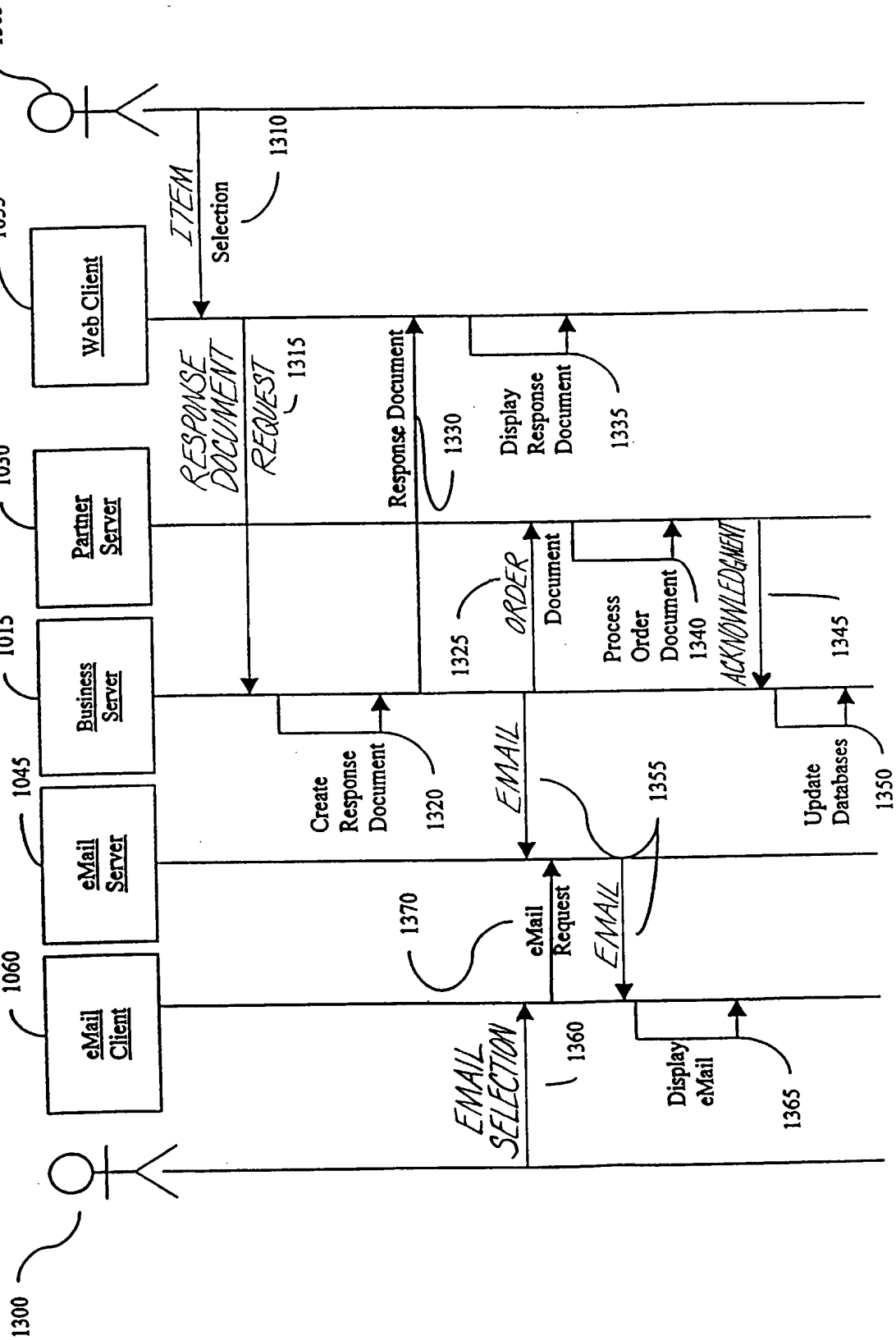

1305
1310
ITEM
Selection
1055
Web Client
1315
RESPONSE DOCUMENT REQUEST
1330
Response Document
1335
Display Response Document
1030
Partner Server
1325
ORDER
Document
1340
Process Order Document
ACKNOWLEDGMENT
1345
1015
Business Server
1320
Create Response Document
EMAIL
1355
Update Databases
1350
1045
eMail Server
1370
eMail Request
EMAIL
1060
eMail Client
EMAIL SELECTION
1360
Display eMail
1365
1300

FIG. 3

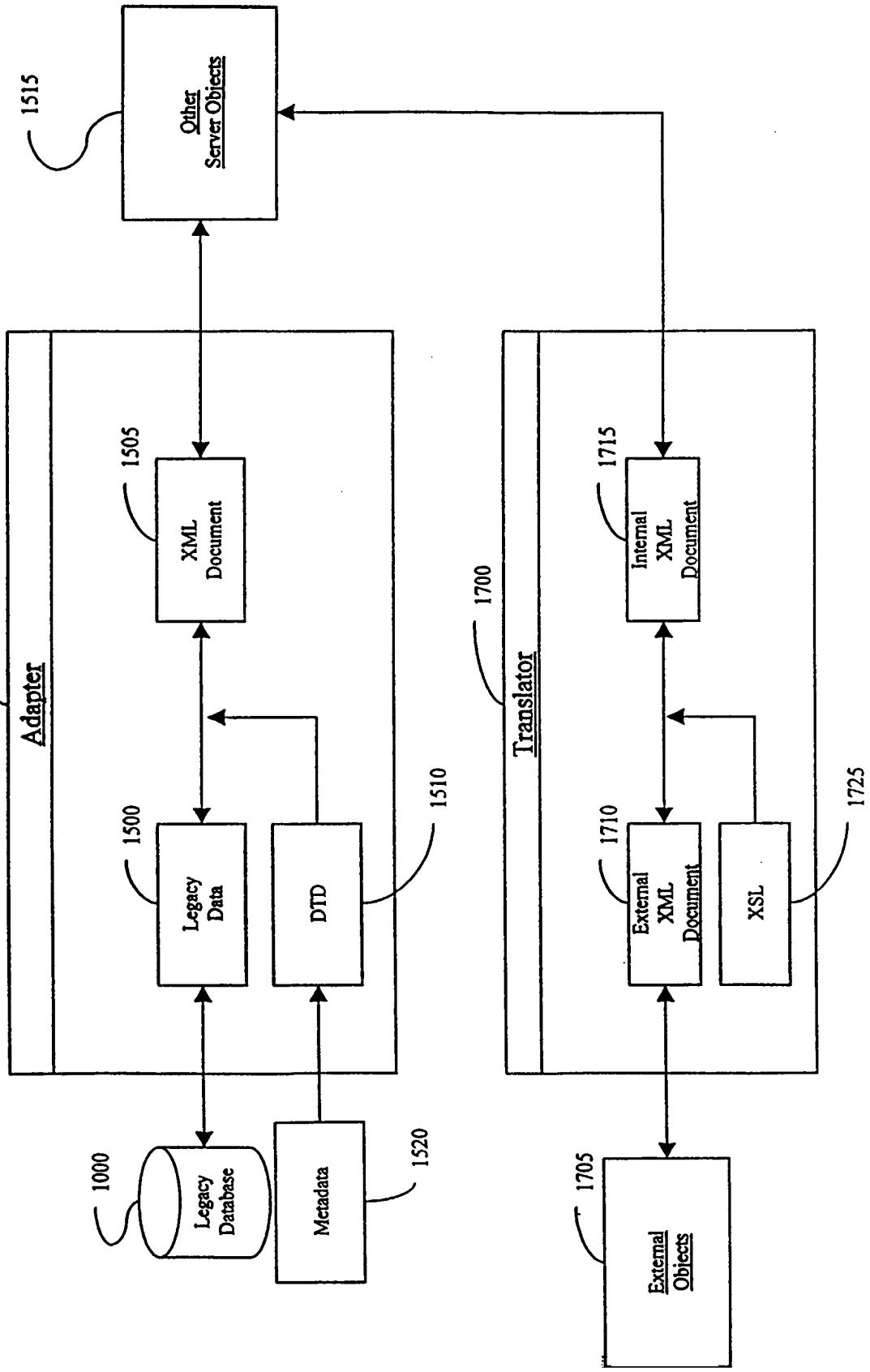

1515
Other Server Objects
1525
Adapter
1505
XML Document
1500
Legacy Data
1510
DTD
1000
Legacy Database
1520
Metadata
1700
Translator
1715
Internal XML Document
1710
External XML Document
1725
XSL
1705
External Objects

FIG. 4

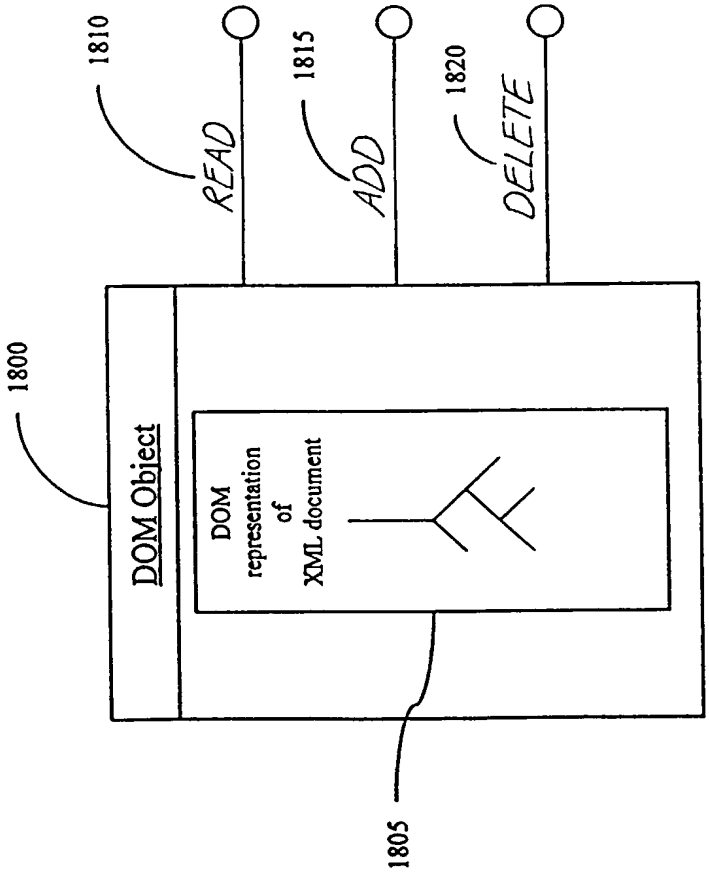
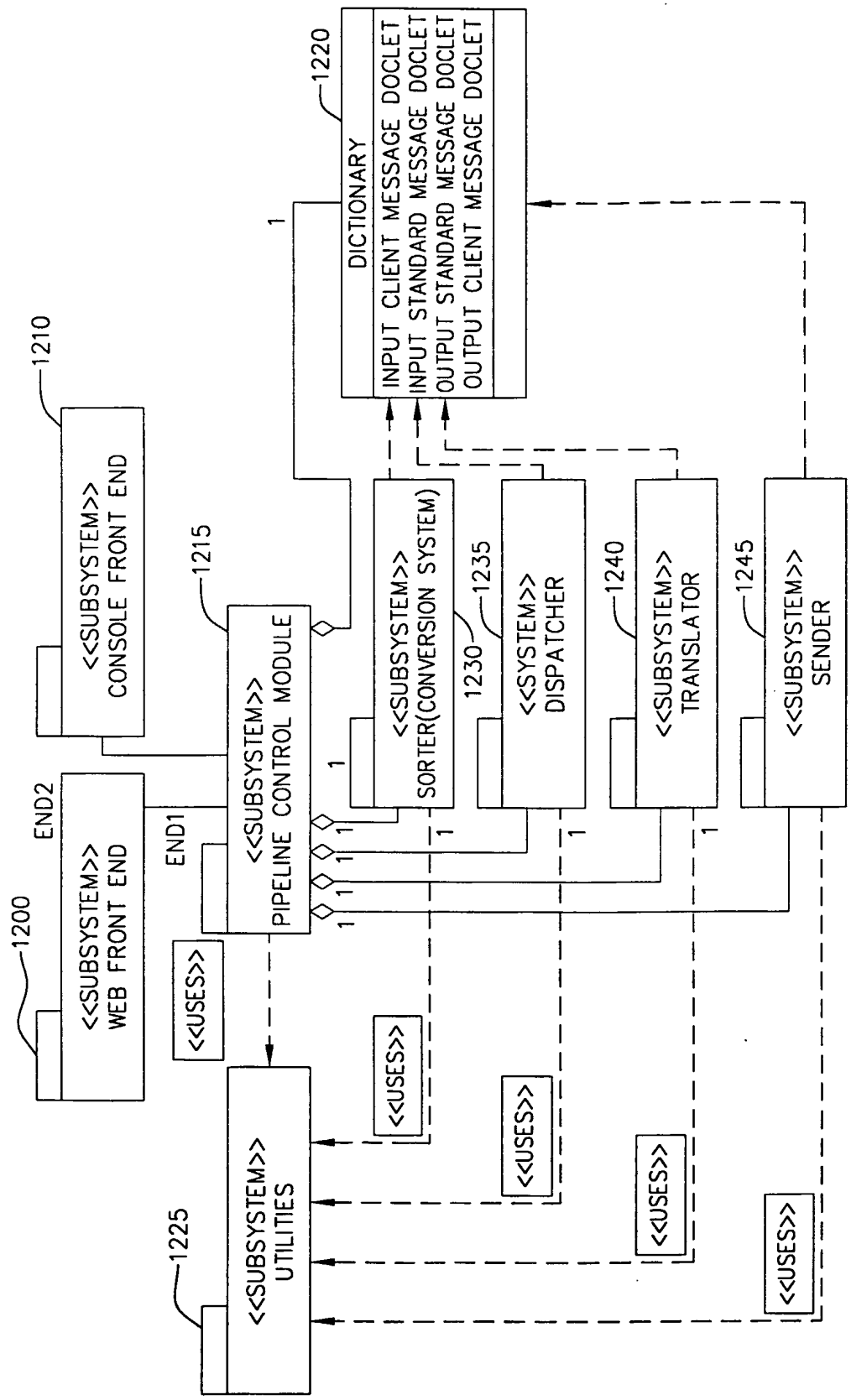

1810
READ
1815
ADD
1820
DELETE
1800
DOM Object
DOM representation of XML document
1805

FIG. 5

FIG.6a


1220
DICTIONARY
INPUT CLIENT MESSAGE DOCLET
INPUT STANDARD MESSAGE DOCLET
OUTPUT STANDARD MESSAGE DOCLET
OUTPUT CLIENT MESSAGE DOCLET
1
1210
<<SUBSYSTEM>>
CONSOLE FRONT END
1215
<<SUBSYSTEM>>
PIPELINE CONTROL MODULE
<<SUBSYSTEM>>
SORTER(CONVERSION SYSTEM)
1230
1235
<<SYSTEM>>
DISPATCHER
1240
<<SUBSYSTEM>>
TRANSLATOR
1245
<<SUBSYSTEM>>
SENDER
END2
END1
1200
<<SUBSYSTEM>>
WEB FRONT END
<<USES>>
<<USES>>
<<USES>>
<<USES>>
<<USES>>
1225
<<SUBSYSTEM>>
UTILITIES

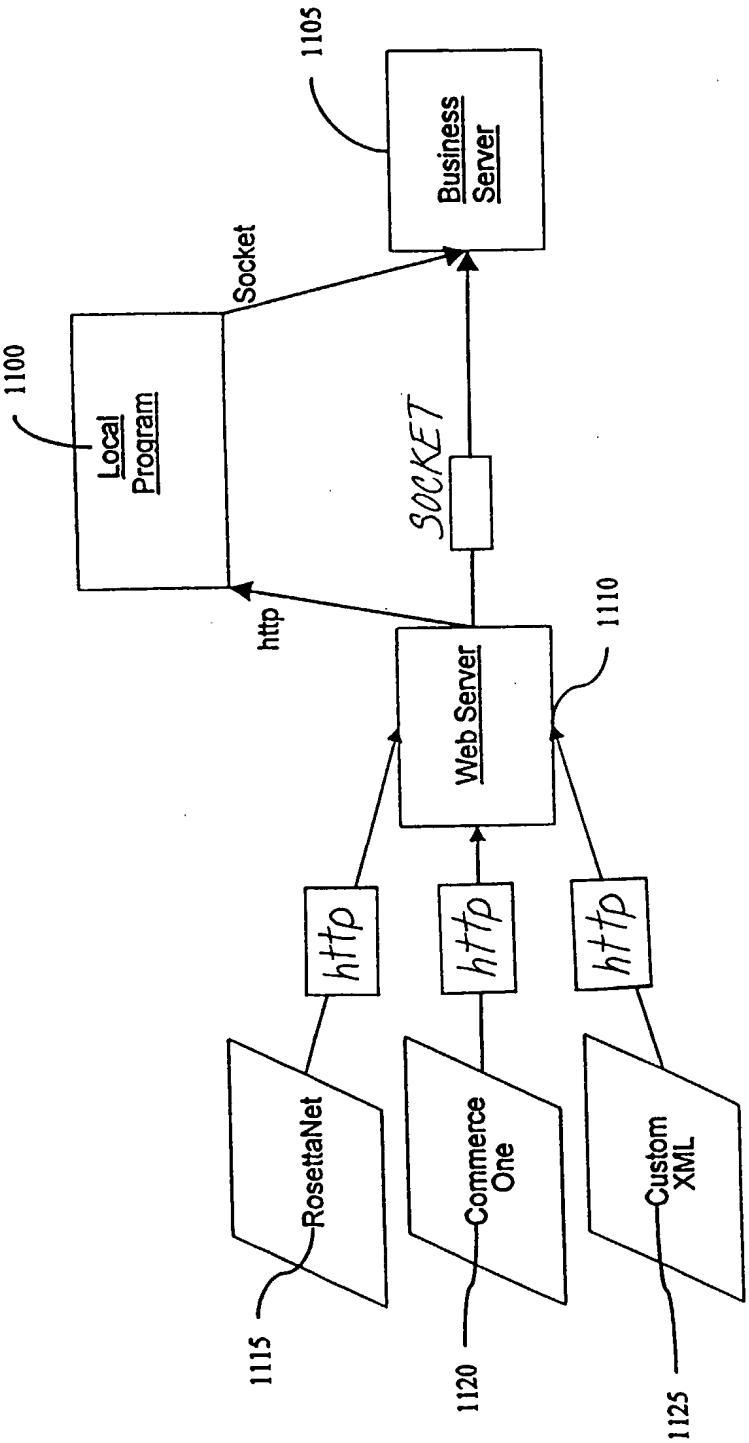

1105
Business Server
Socket
1100
Local Program
SOCKET
http
Web Server
1110
http
http
http
RosettaNet
Commerce One
Custom XML
1115
1120
1125

FIG. 6b

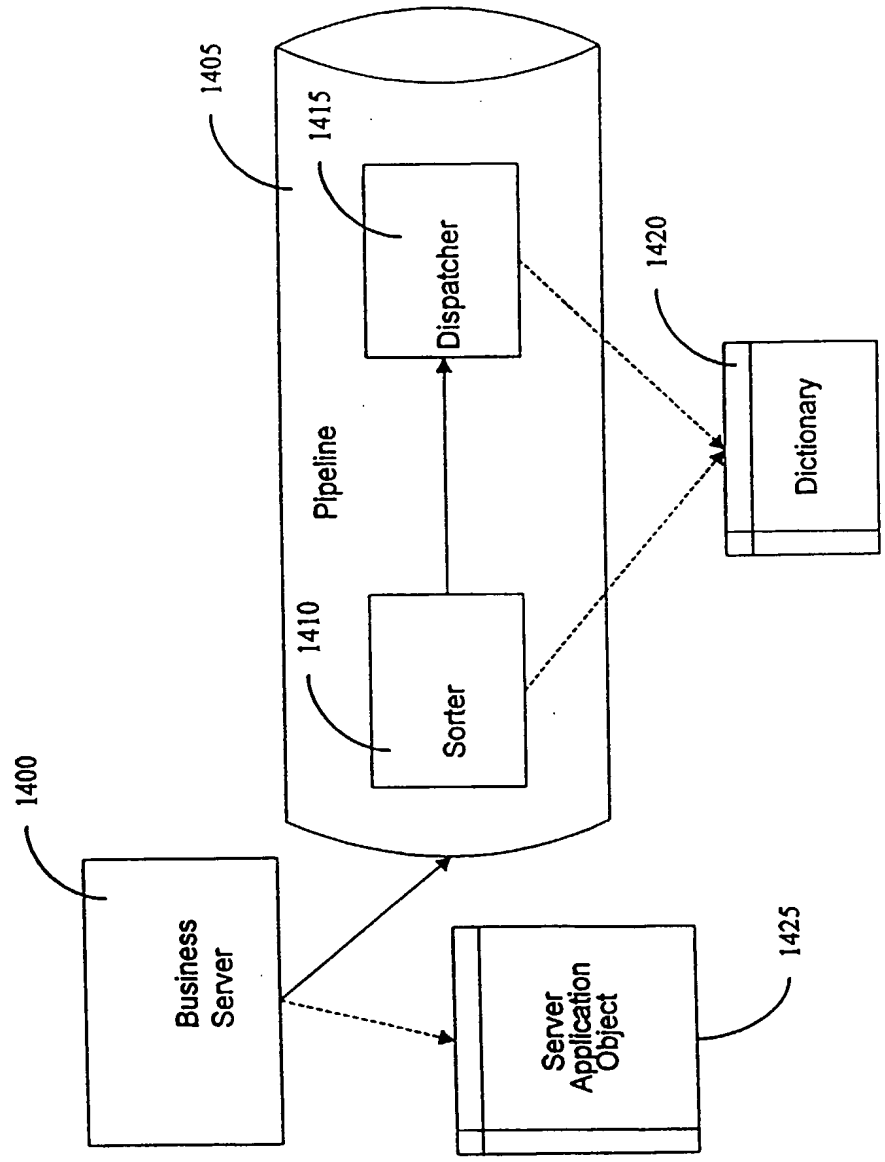

1405
1415
Dispatcher
1420
Dictionary
Pipeline
1410
Sorter
1400
Business Server
Server Application Object
1425

FIG. 6c

FIG. 6d

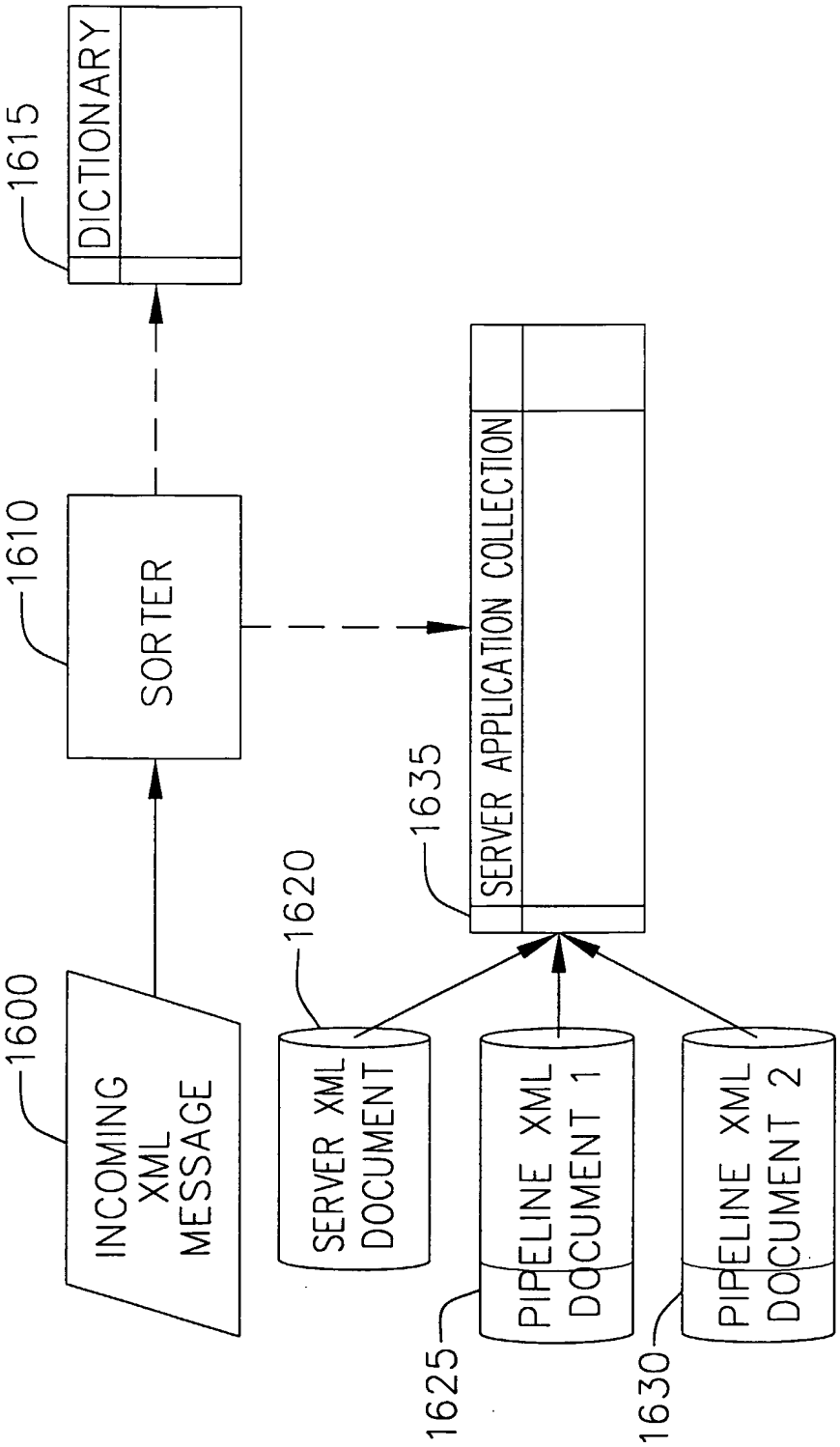

1615
DICTIONARY
1610
SORTER
1635
SERVER APPLICATION COLLECTION
1600
INCOMING XML MESSAGE
1620
SERVER XML DOCUMENT
PIPELINE XML DOCUMENT 1
PIPELINE XML DOCUMENT 2
1625
1630

FIG. 6e

```
1900
<BusinessServer>
    1910                1915
    <BusinessProcess name='ABCPurchaseOrder'                1905
                pipeline='plABCPurchaseOrder.xml'           1920
                translator='ABCPurchaseOrder.xsl'>
1930    <Criteria>
            <TestBatch>                     1935            1925
1950            <Test XPath='/MyRoot/[@id='x']foo/...' />
            ...
            </TestBatch>
        ...
        </Criteria>
    </BusinessProcess>
1960    <BusinessProcess name='TrickyPurchaseOrder'
                pipeline='plTrickyPurchaseOrder.xml'
                translator='TrickyPurchaseOrder.xsl'
1955        <Object>                                        1945
                OldStyleSorter.COMObjectName
            </Object>
    </BusinessProcess>
    ...
</BusinessServer>
```

```
<Pipeline>                                                    2100
  <ExceptionHandler handler='ADODB.ExceptionHandler'>         2105
    <Sender>                                                  2107
      ADODB.ExceptionSenderHTTPResponse                       2110
    </Sender>                                                 2115
    <Sender>
      ADODB.ExceptionSenderEMail                              2120
    </Sender>
    ...
  </ExceptionHandler>                                         2125
  <Component type='DBDoclet' target='MyDoclet'                2130
    ABCPO_DB_ReadInventory.xml                                2135
  </Component>                                                2136
  <Component type='Rule'>                                     2145
    ABCPO.MyCOMObject                                         2146
  </Component>                                                2150
  <Component type='Transform' src='MyDoclet                   2155
    target='MyResponse'>                                      2160
    ABCPO_Rosettanet-xCBL.xsl                                 2165
  </Component>                                                2170
  <Component type='Sender' protocol='HTTPResponse'            2175
    src='MyResponse' />                                       2180
  ...                                                         2185
</Pipeline>                                                   2190
```

FIG. 6f

FIG. 7

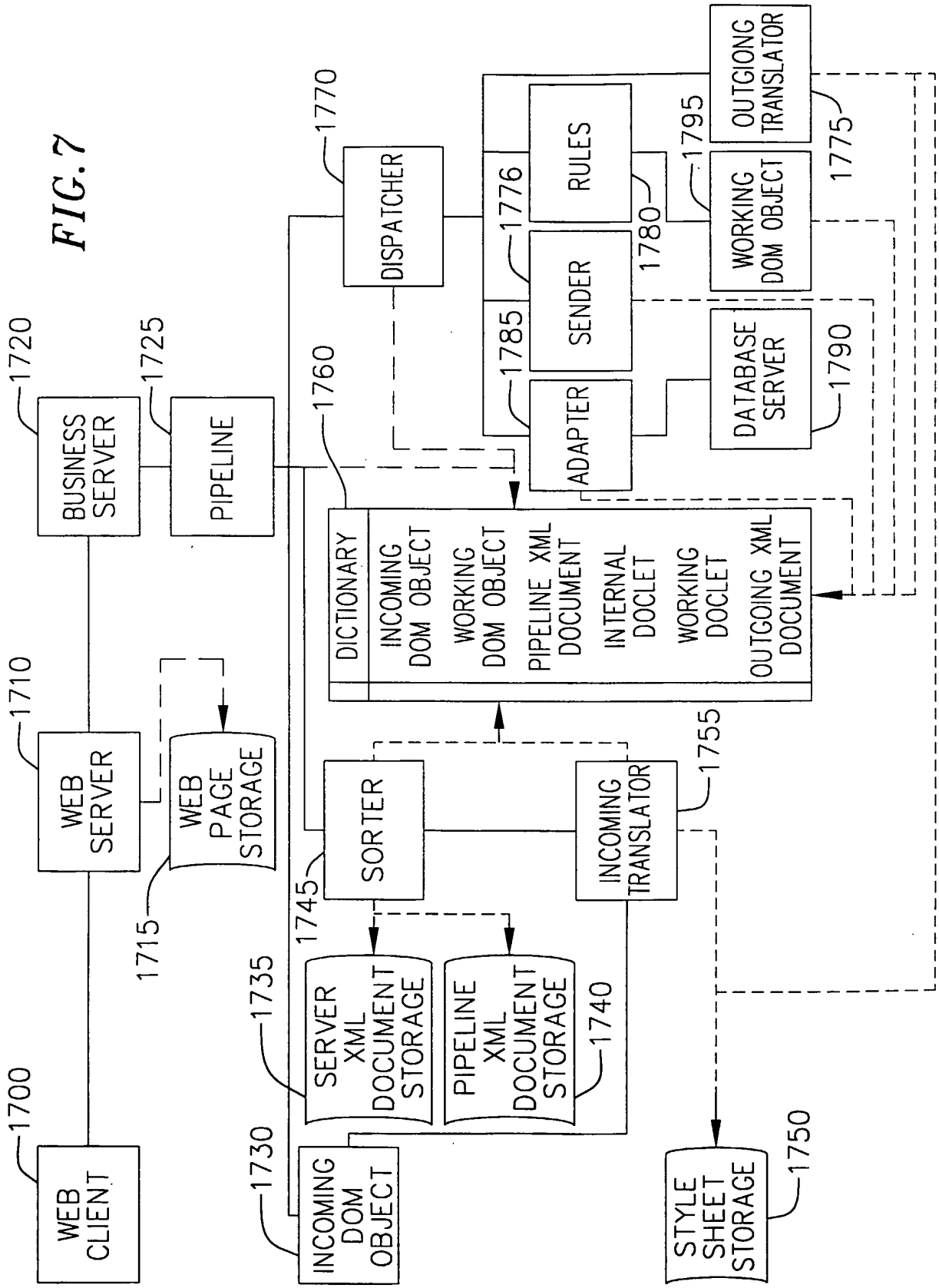

1700
WEB CLIENT
1710
WEB SERVER
1715
WEB PAGE STORAGE
1720
BUSINESS SERVER
1725
PIPELINE
1730
INCOMING DOM OBJECT
1735
SERVER XML DOCUMENT STORAGE
1740
PIPELINE XML DOCUMENT STORAGE
1745
SORTER
1750
STYLE SHEET STORAGE
1755
INCOMING TRANSLATOR
1760
DICTIONARY
INCOMING DOM OBJECT
WORKING DOM OBJECT
PIPELINE XML DOCUMENT
INTERNAL DOCLET
WORKING DOCLET
OUTGOING XML DOCUMENT
1770
DISPATCHER
1775
OUTGIONG TRANSLATOR
1776
RULES
1780
SENDER
1785
ADAPTER
1790
DATABASE SERVER
1795
WORKING DOM OBJECT

FIG. 8

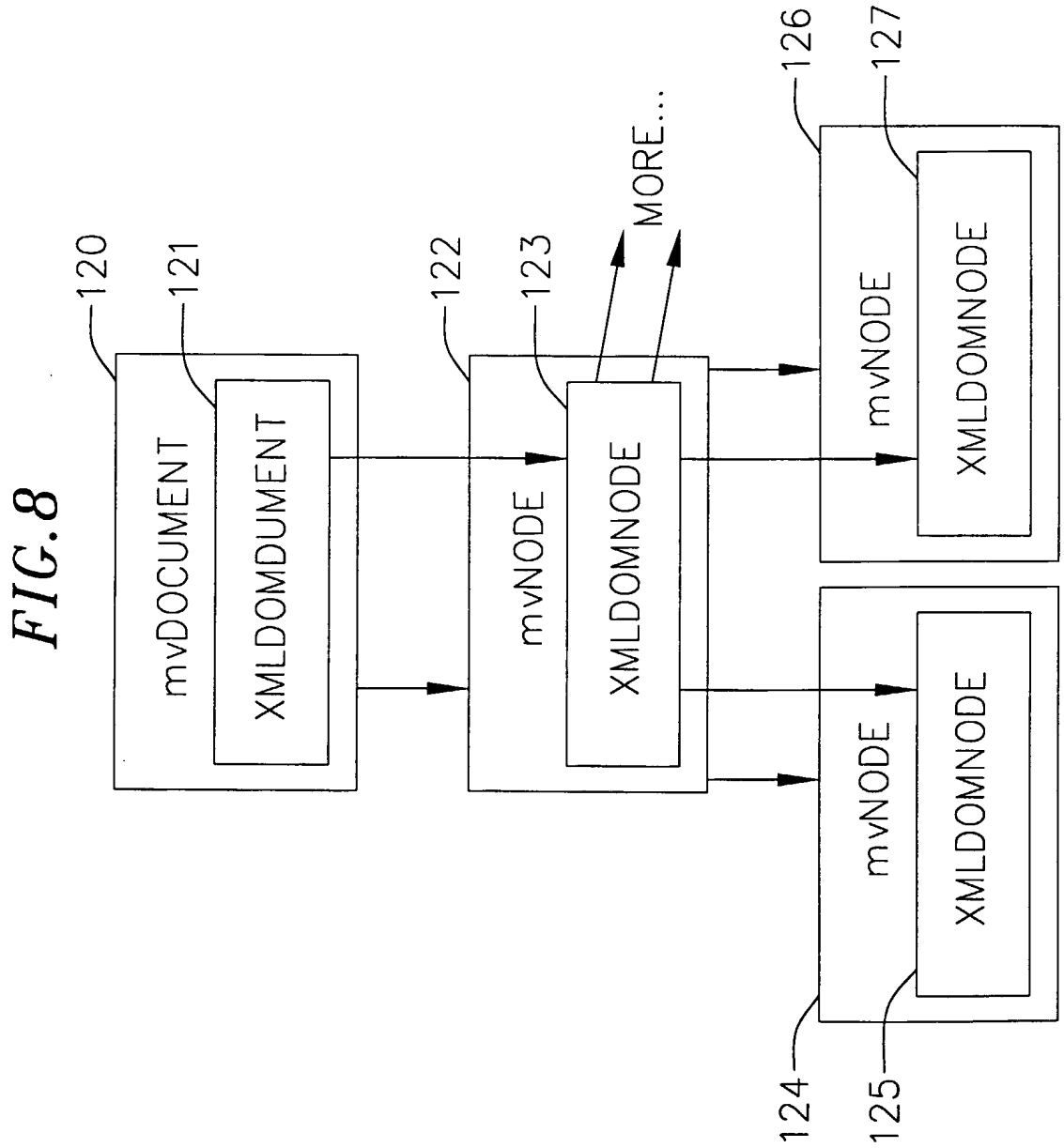
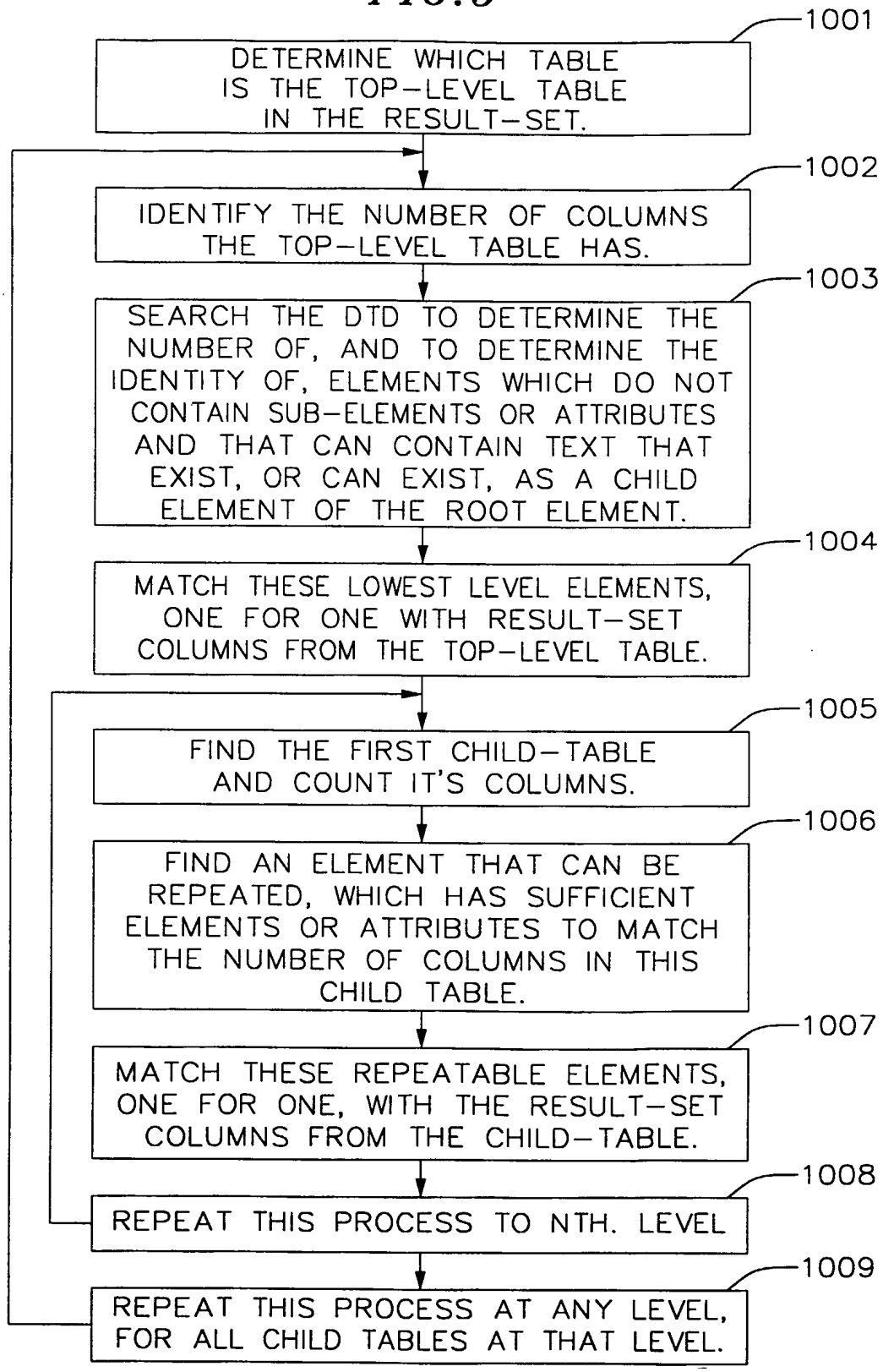

120
121
122
123
126
127
124
125
mvDOCUMENT
XMLDOMDUMENT
mvNODE
XMLDOMNODE
MORE...
mvNODE
XMLDOMNODE
mvNODE
XMLDOMNODE

FIG. 9


1001
DETERMINE WHICH TABLE IS THE TOP-LEVEL TABLE IN THE RESULT-SET.
1002
IDENTIFY THE NUMBER OF COLUMNS THE TOP-LEVEL TABLE HAS.
1003
SEARCH THE DTD TO DETERMINE THE NUMBER OF, AND TO DETERMINE THE IDENTITY OF, ELEMENTS WHICH DO NOT CONTAIN SUB-ELEMENTS OR ATTRIBUTES AND THAT CAN CONTAIN TEXT THAT EXIST, OR CAN EXIST, AS A CHILD ELEMENT OF THE ROOT ELEMENT.
1004
MATCH THESE LOWEST LEVEL ELEMENTS, ONE FOR ONE WITH RESULT-SET COLUMNS FROM THE TOP-LEVEL TABLE.
1005
FIND THE FIRST CHILD-TABLE AND COUNT IT'S COLUMNS.
1006
FIND AN ELEMENT THAT CAN BE REPEATED, WHICH HAS SUFFICIENT ELEMENTS OR ATTRIBUTES TO MATCH THE NUMBER OF COLUMNS IN THIS CHILD TABLE.
1007
MATCH THESE REPEATABLE ELEMENTS, ONE FOR ONE, WITH THE RESULT-SET COLUMNS FROM THE CHILD-TABLE.
1008
REPEAT THIS PROCESS TO NTH. LEVEL
1009
REPEAT THIS PROCESS AT ANY LEVEL, FOR ALL CHILD TABLES AT THAT LEVEL.

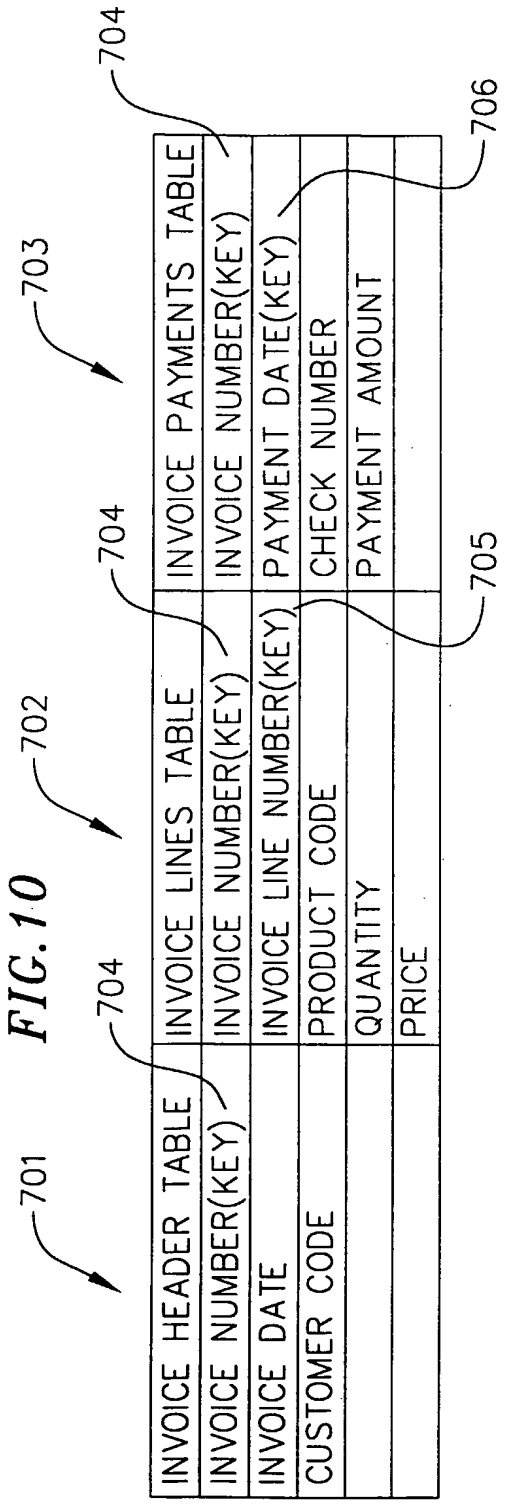

701
704
702
704
703
704
705
706
INVOICE HEADER TABLE
INVOICE NUMBER(KEY)
INVOICE DATE
CUSTOMER CODE
INVOICE LINES TABLE
INVOICE NUMBER(KEY)
INVOICE LINE NUMBER(KEY)
PRODUCT CODE
QUANTITY
PRICE
INVOICE PAYMENTS TABLE
INVOICE NUMBER(KEY)
PAYMENT DATE(KEY)
CHECK NUMBER
PAYMENT AMOUNT

FIG. 10

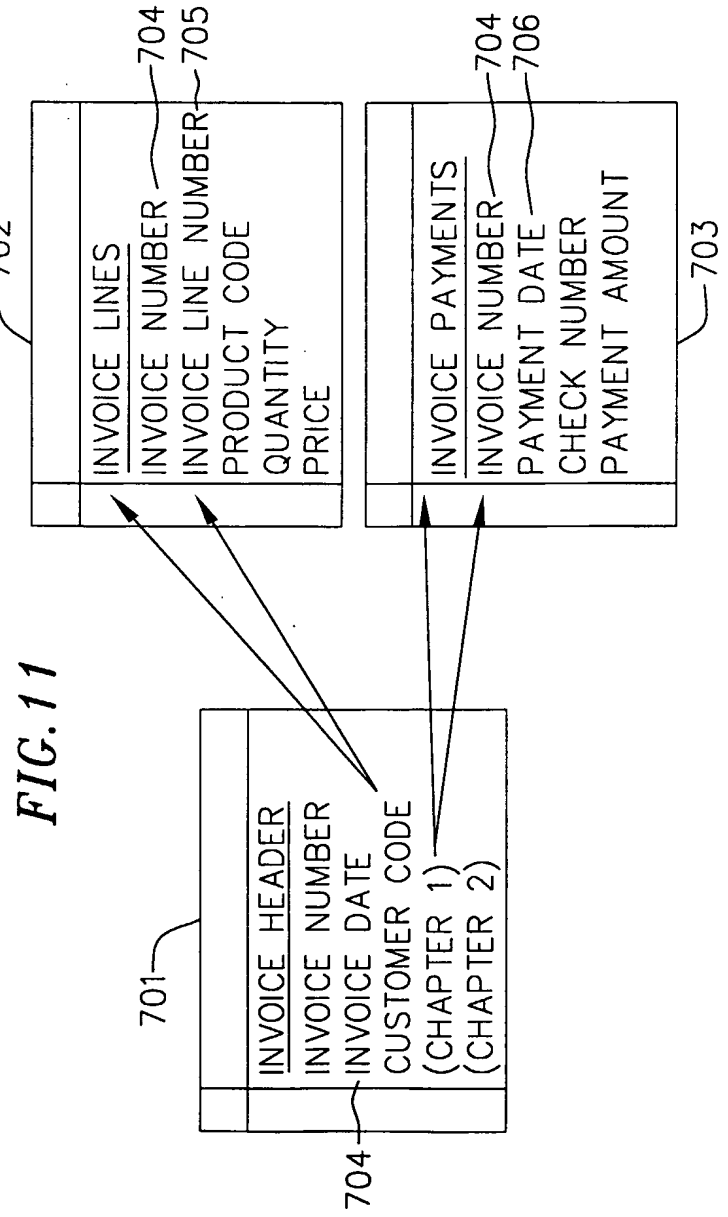

701
704
INVOICE HEADER
INVOICE NUMBER
INVOICE DATE
CUSTOMER CODE
(CHAPTER 1)
(CHAPTER 2)
702
704
705
INVOICE LINES
INVOICE NUMBER
INVOICE LINE NUMBER
PRODUCT CODE
QUANTITY
PRICE
704
706
703
INVOICE PAYMENTS
INVOICE NUMBER
PAYMENT DATE
CHECK NUMBER
PAYMENT AMOUNT

FIG. 11

FIG. 12

```
<INVOICEHEADER>
    <!--ALL ELEMENTS OF THE INVOICEHEADER TABLE GO IN AT THIS
        LEVEL -->
    <INVOICENUMBER>
        ...
    </INVOICENUMBER>
    <INVOICEDATA>
        ...
    </INVOICEDATE>
    <CUSTOMERNUMBER>
        ...
    </CUSTOMERNUMBER>
    <INVOICELINES>
        <!-- ALL ELEMENTS OF THE INVOICELINES TABLE GO IN AT THIS
            LEVEL -->
    </INVOICELINES>
    <!-- REPEAT THE INVOICELINES ELEMENTS FOR EACH INVOICELINES ROW
    THAT GOES WITH THE INVOICEHEADER RECORD -->
    <INVOICEPAYMENTS>
        <!-- ALL ELEMENTS OF THE INVOICEPAYMENTS TABLE GO IN AT
            THIS LEVEL -->
    </INVOICEPAYMENTS>
    <!-- REPEAT THE INVOICEPAYMENTS ELEMENTS FOR EACH INVOICEPAYMENTS
        ROW THAT GOES WITH THE INVOICEHEADER RECORD -->
</INVOICEHEADER>
```

FIG. 13

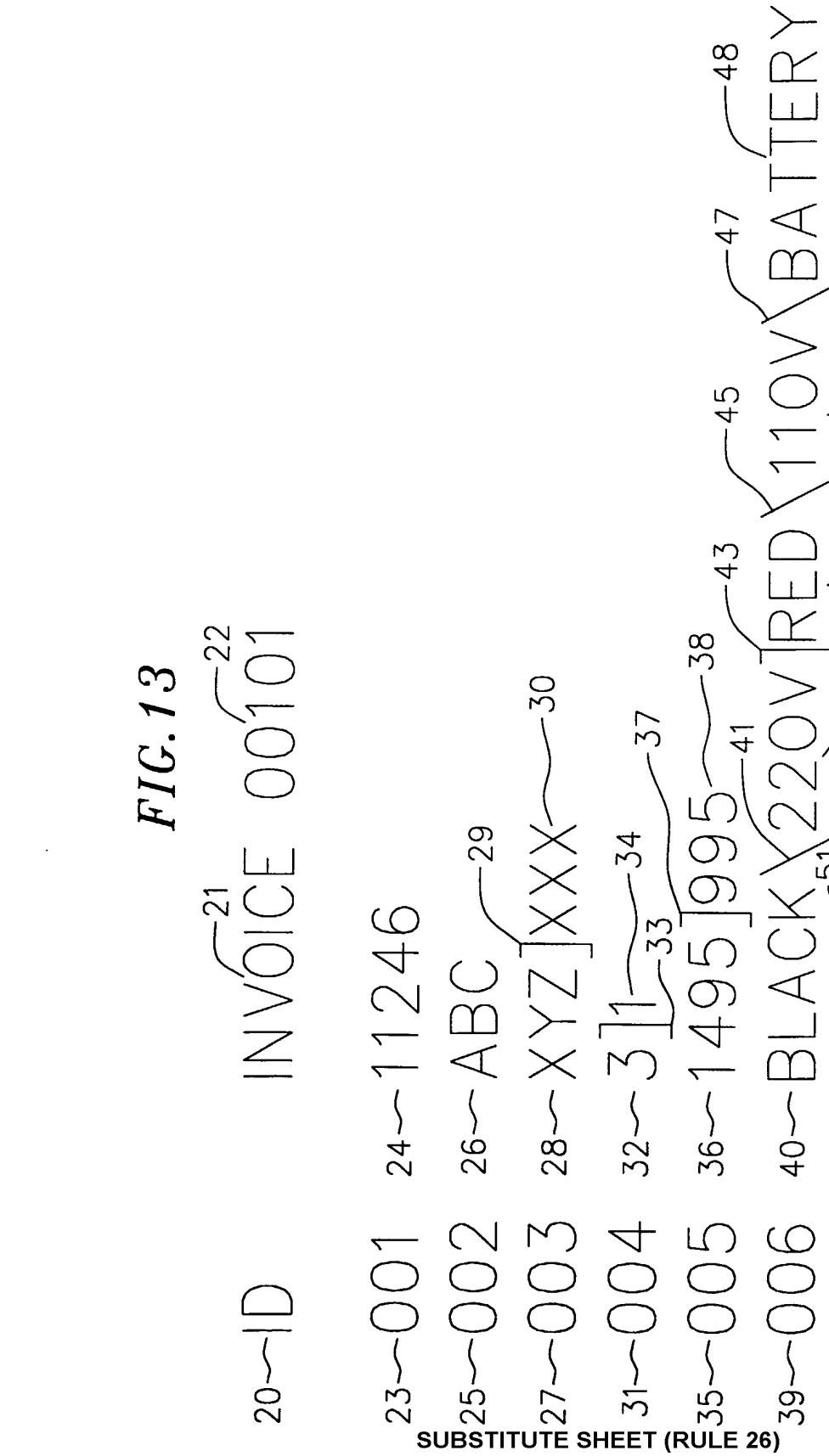

20 ID
21 INVOICE
22 00101
23 001
24 11246
25 002
26 ABC
27 003
28 XYZ
29
30 XXX
31 004
32 3
33
34
35 005
36 1495
37
38 995
39 006
40 BLACK
41
42 220V
43 RED
44
45 110V
46
47 BATTERY
48
49 007
50 1023
51
52 1076
53 008
54 11243
55
56 11267
57 009
58 1000
59 1480
60

FIG. 14

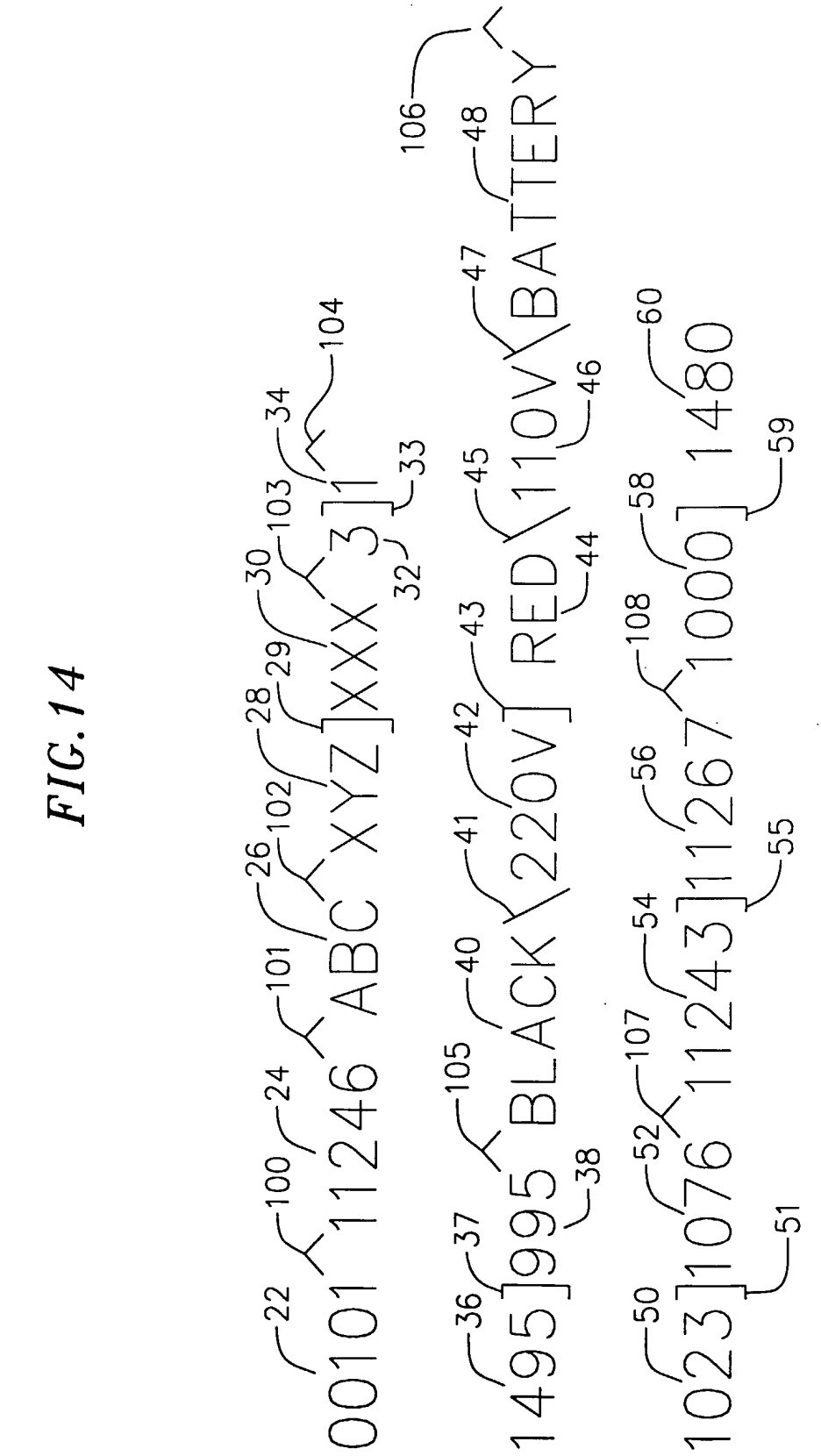

00101 11246 ABC XYZ XXX 3 1
1495 995 BLACK 220V RED 110V BATTERY
1023 10076 11243 11267 1000 1480
22
100
24
101
26
102
28
29
30
103
34
104
33
32
106
36
37
105
40
41
42
43
44
45
46
47
48
38
50
52
107
54
56
108
58
60
59
55
51

FIG. 15

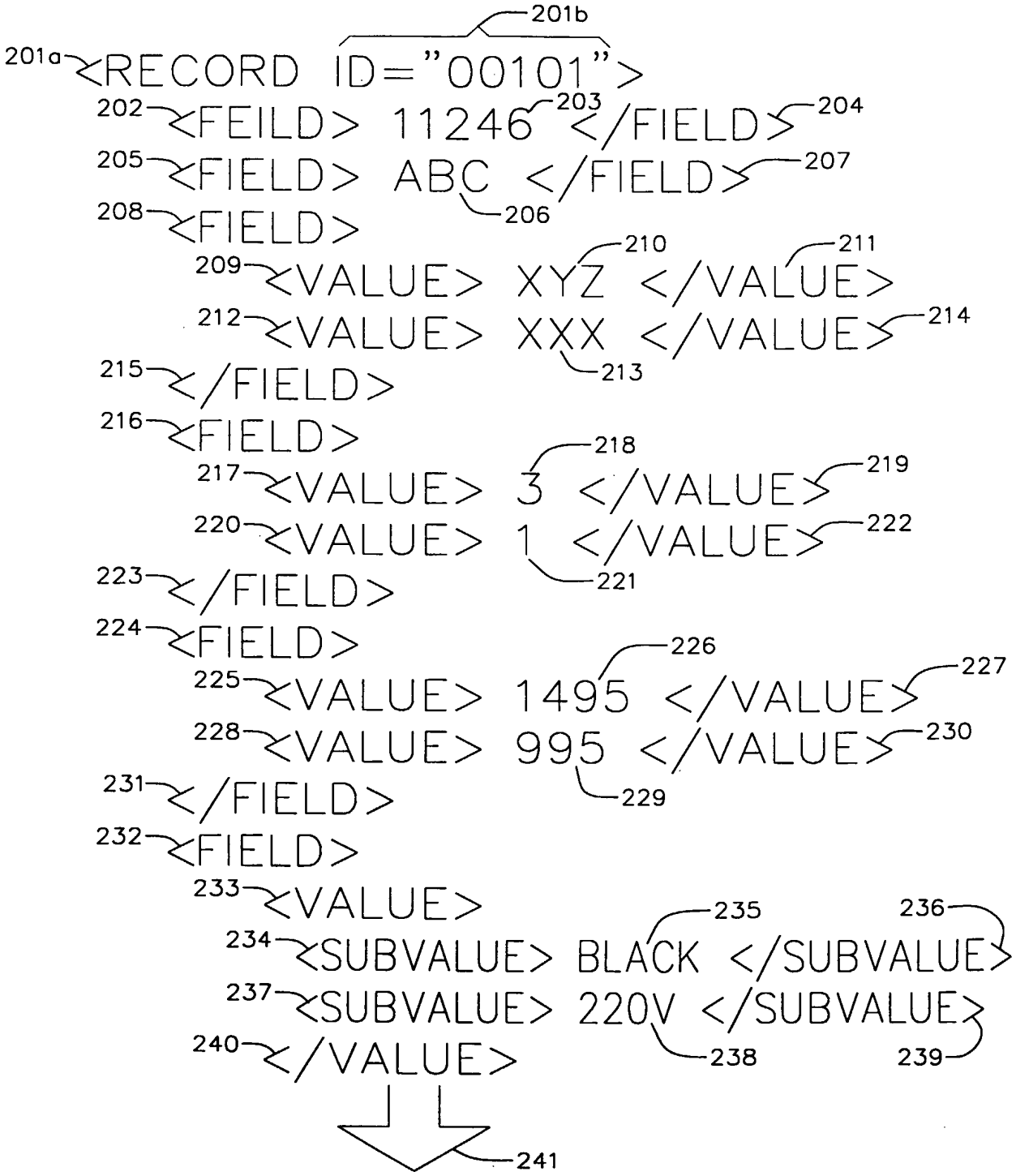

201a
201b
<RECORD ID="00101">
202
<FEILD> 11246 203 </FIELD> 204
205
<FIELD> ABC </FIELD> 207
206
208
<FIELD>
209
<VALUE> XYZ 210 </VALUE> 211
212
<VALUE> XXX </VALUE> 214
213
215
</FIELD>
216
<FIELD>
217
<VALUE> 3 218 </VALUE> 219
220
<VALUE> 1 </VALUE> 222
221
223
</FIELD>
224
<FIELD>
225
<VALUE> 1495 226 </VALUE> 227
228
<VALUE> 995 </VALUE> 230
229
231
</FIELD>
232
<FIELD>
233
<VALUE>
234
<SUBVALUE> BLACK 235 </SUBVALUE> 236
237
<SUBVALUE> 220V </SUBVALUE>
238
239
240
</VALUE>
241

FIG. 16

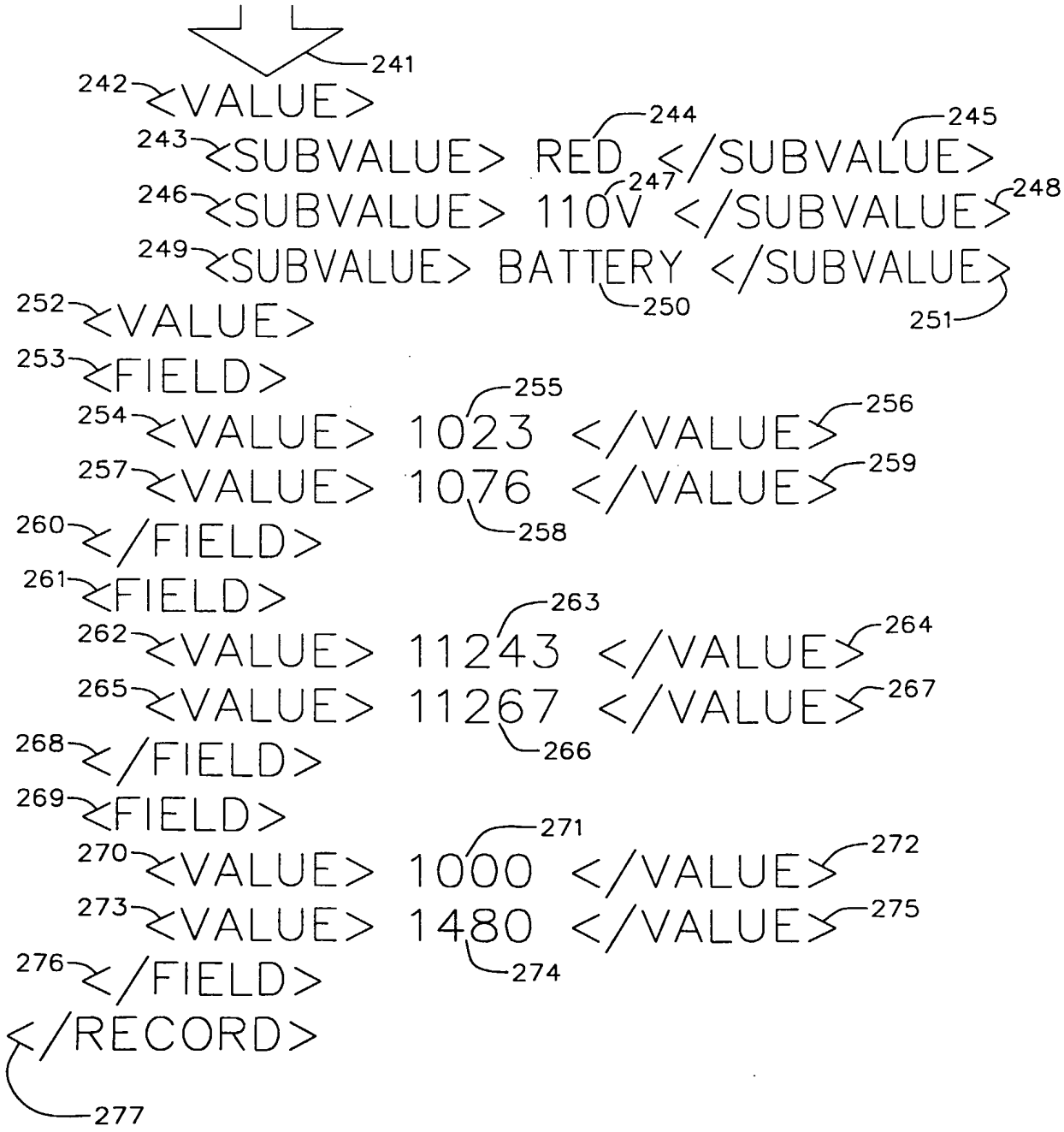

241
242 <VALUE>
243 <SUBVALUE> RED </SUBVALUE>
244
245
246 <SUBVALUE> 110V </SUBVALUE>
247
248
249 <SUBVALUE> BATTERY </SUBVALUE>
250
251
252 <VALUE>
253 <FIELD>
254 <VALUE> 1023 </VALUE>
255
256
257 <VALUE> 1076 </VALUE>
258
259
260 </FIELD>
261 <FIELD>
262 <VALUE> 11243 </VALUE>
263
264
265 <VALUE> 11267 </VALUE>
266
267
268 </FIELD>
269 <FIELD>
270 <VALUE> 1000 </VALUE>
271
272
273 <VALUE> 1480 </VALUE>
274
275
276 </FIELD>
</RECORD>
277

FIG. 17

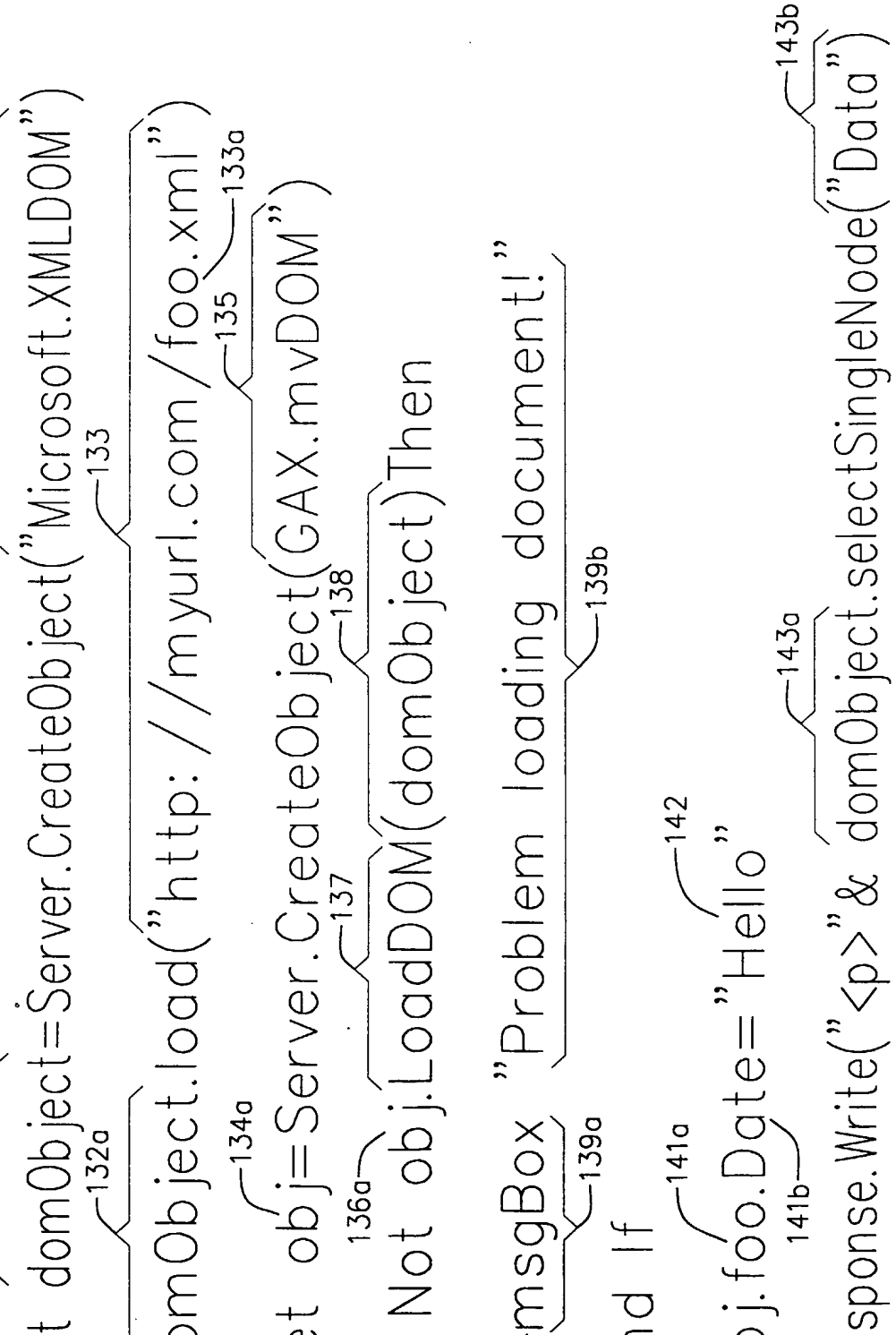
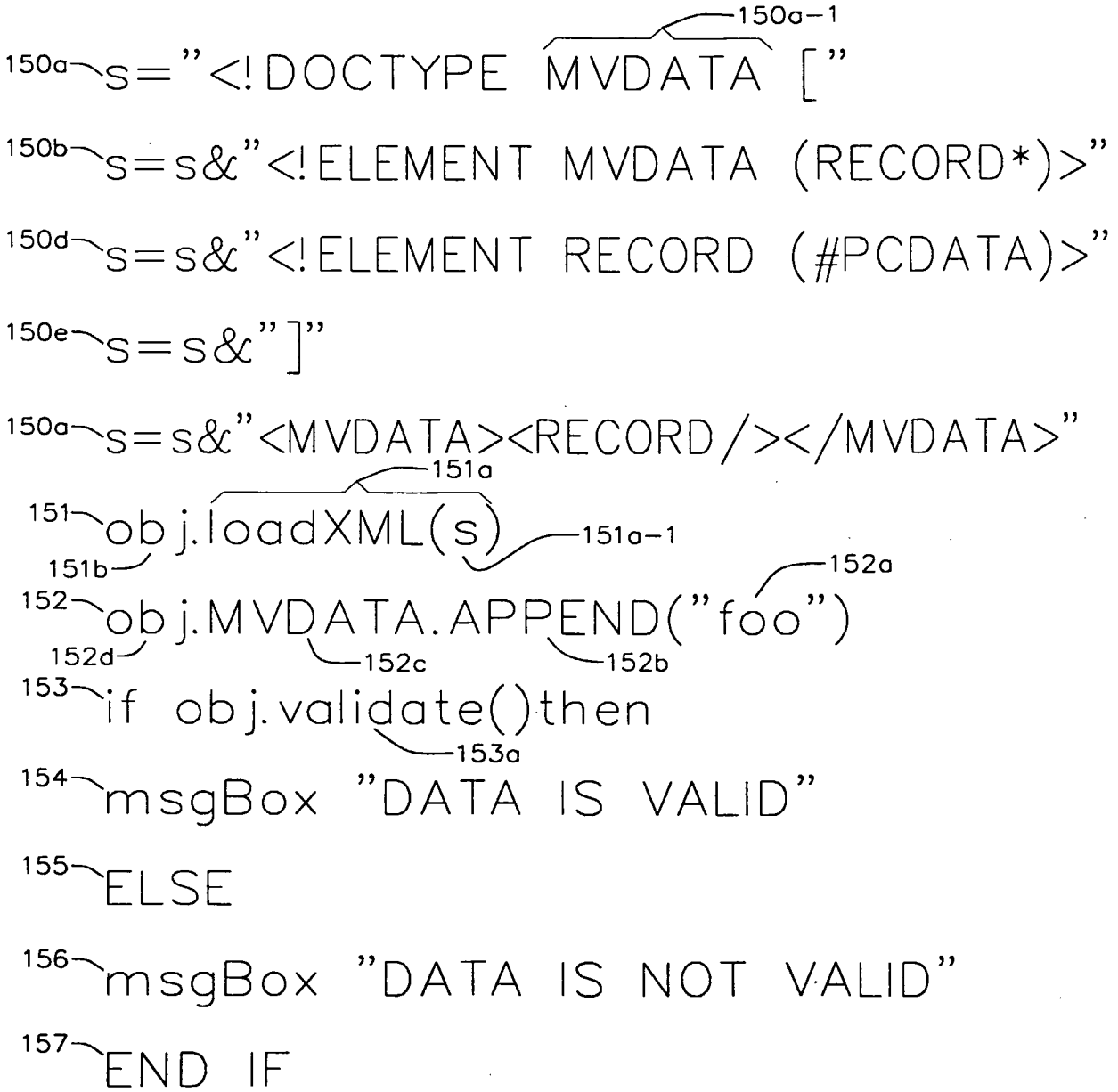

130 set domObject=Server.CreateObject("Microsoft.XMLDOM")
131
130a
132 domObject.load("http://myurl.com/foo.xml")
133
132a
134 set obj=Server.CreateObject(GAX.mvDOM")
135
133a
134a
136 If Not obj.LoadDOM(domObject)Then
137
138
136a
139 msgBox "Problem loading document!"
139b
139a
140 End If
141a
141 obj.foo.Date="Hello"
142
141b
143a
143 Response.Write("<p>"& domObject.selectSingleNode("Data")
143b
value & "world")
143c

FIG. 18


150a-1
150a s="<!DOCTYPE MVDATA [">
150b s=s&"<!ELEMENT MVDATA (RECORD*)>"
150d s=s&"<!ELEMENT RECORD (#PCDATA)>"
150e s=s&"]>"
150a s=s&"<MVDATA><RECORD/></MVDATA>"
151a
151 obj.loadXML(s)
151a-1
151b
152a
152 obj.MVDATA.APPEND("foo")
152d
152c
152b
153 if obj.validate()then
153a
154 msgBox "DATA IS VALID"
155 ELSE
156 msgBox "DATA IS NOT VALID"
157 END IF

160a

```
<MVDATA>
    <RECORD>
      <FIELD>
        I'M FIRST
      </FIELD>
      <FIELD>
        NEW DATA
      </FIELD>
    </RECORD>
</MVDATA>
```

FIG. 19

FIG. 20

160a 160b 160c 160d

```
<MVDATA XML:SPACE="PRESERVE">
161  <RECORD>
       <FIELD>                    167
         I'M FIRST                168
       </FIELD>                   169
       <FIELD>                    162
         NEW DATA                 163
       </FIELD>                   164
    </RECORD>                     165
</MVDATA>                         166
```

FIG.21

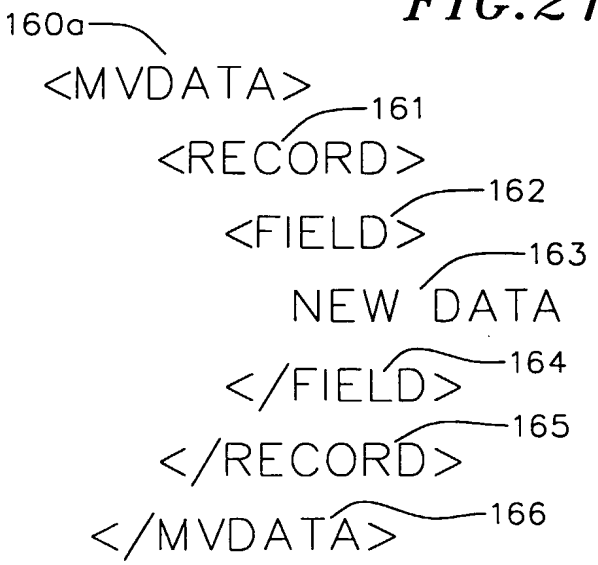

160a
<MVDATA>
161
<RECORD>
162
<FIELD>
163
NEW DATA
</FIELD>
164
</RECORD>
165
</MVDATA>
166

FIG.22

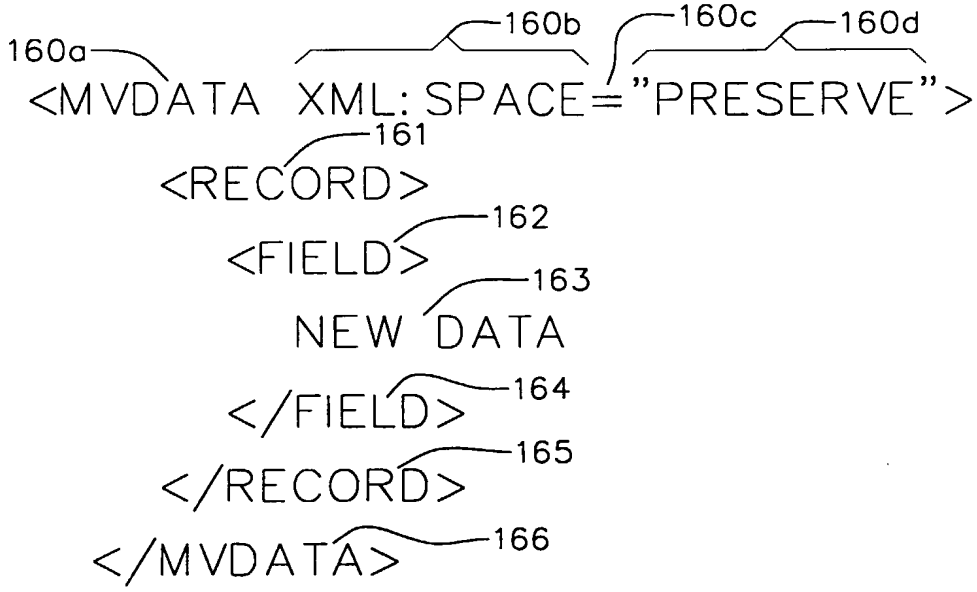

160a
160b
160c
160d
<MVDATA XML:SPACE="PRESERVE">
161
<RECORD>
162
<FIELD>
163
NEW DATA
</FIELD>
164
</RECORD>
165
</MVDATA>
166

FIG.23

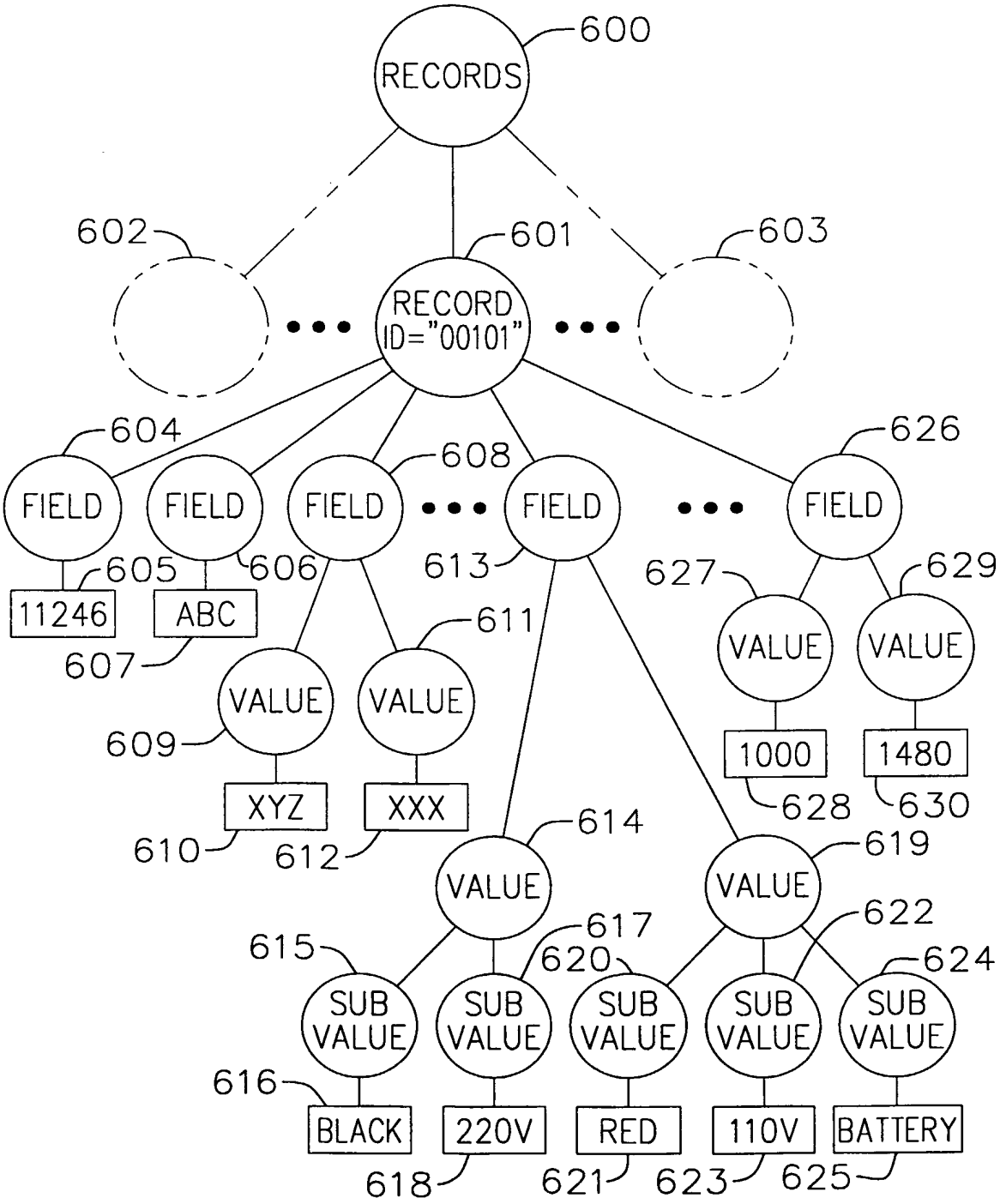

RECORDS
600
602
601
603
RECORD
ID="00101"
604
608
626
FIELD
FIELD
FIELD
FIELD
FIELD
605
606
613
627
629
11246
ABC
611
VALUE
VALUE
607
609
1000
1480
XYZ
XXX
614
619
610
612
628
630
VALUE
VALUE
615
617
620
622
624
SUB VALUE
SUB VALUE
SUB VALUE
SUB VALUE
SUB VALUE
616
BLACK
220V
RED
110V
BATTERY
618
621
623
625

FIG.24

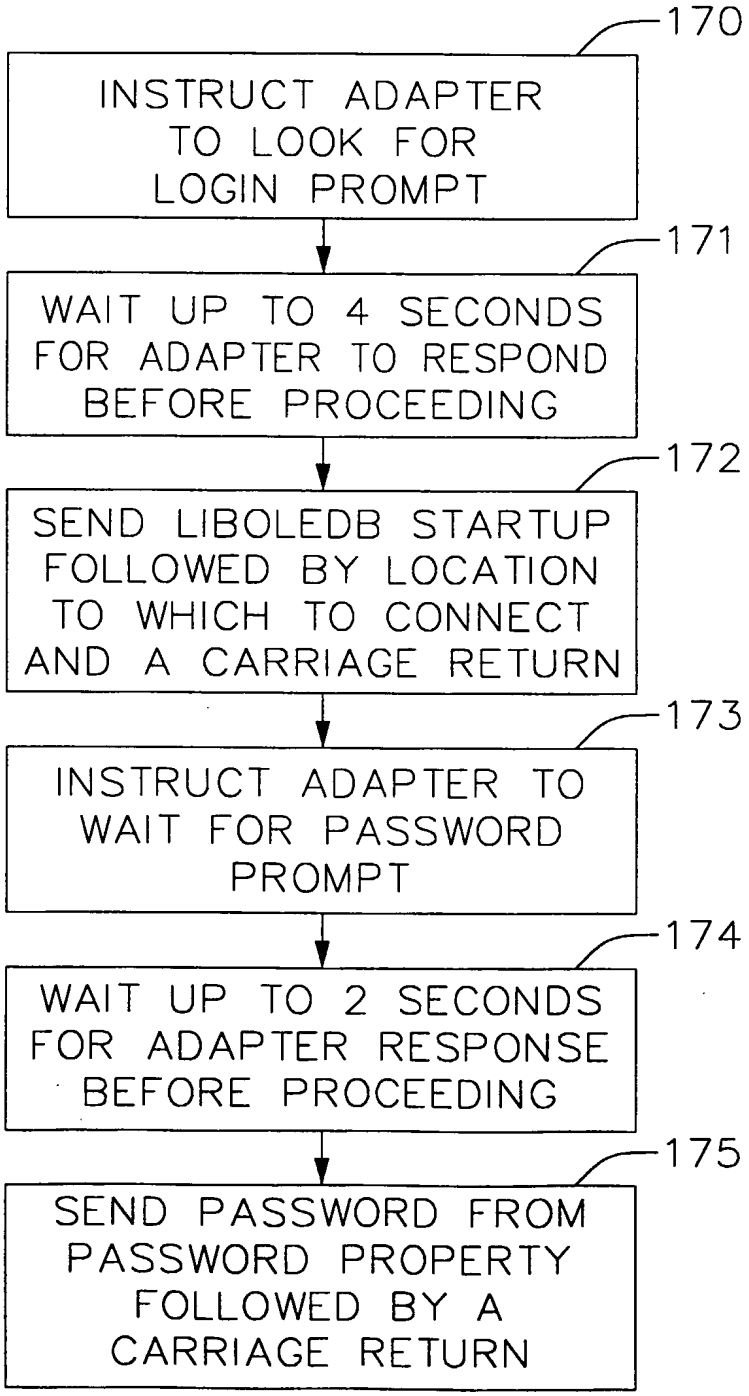

170
INSTRUCT ADAPTER TO LOOK FOR LOGIN PROMPT
171
WAIT UP TO 4 SECONDS FOR ADAPTER TO RESPOND BEFORE PROCEEDING
172
SEND LIBOLEDB STARTUP FOLLOWED BY LOCATION TO WHICH TO CONNECT AND A CARRIAGE RETURN
173
INSTRUCT ADAPTER TO WAIT FOR PASSWORD PROMPT
174
WAIT UP TO 2 SECONDS FOR ADAPTER RESPONSE BEFORE PROCEEDING
175
SEND PASSWORD FROM PASSWORD PROPERTY FOLLOWED BY A CARRIAGE RETURN

FIG.25

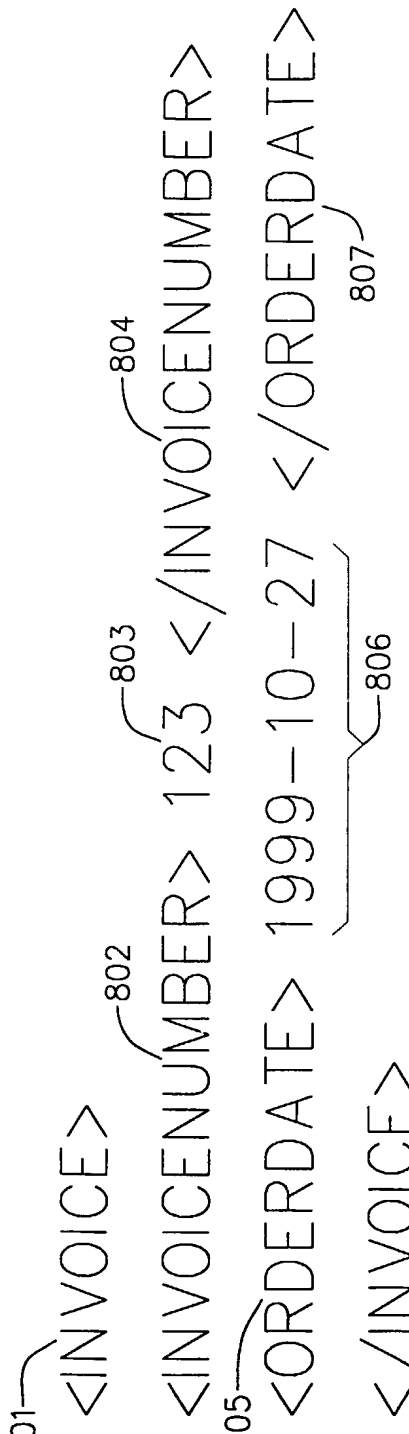

801
<INVOICE>
802
<INVOICENUMBER> 123 </INVOICENUMBER>
804
803
805
<ORDERDATE> 1999-10-27 </ORDERDATE>
807
806
</INVOICE>
808

FIG.26


811
812
OBJ!LOADXML(
814
"
815
816
INVOICE!XML
817
818
"
819
820
STRING DATE=OBJ!INVOICE,ORDERDATE!TEXT
821
822
823
824
811
816
825
805
826
827

FIG.27

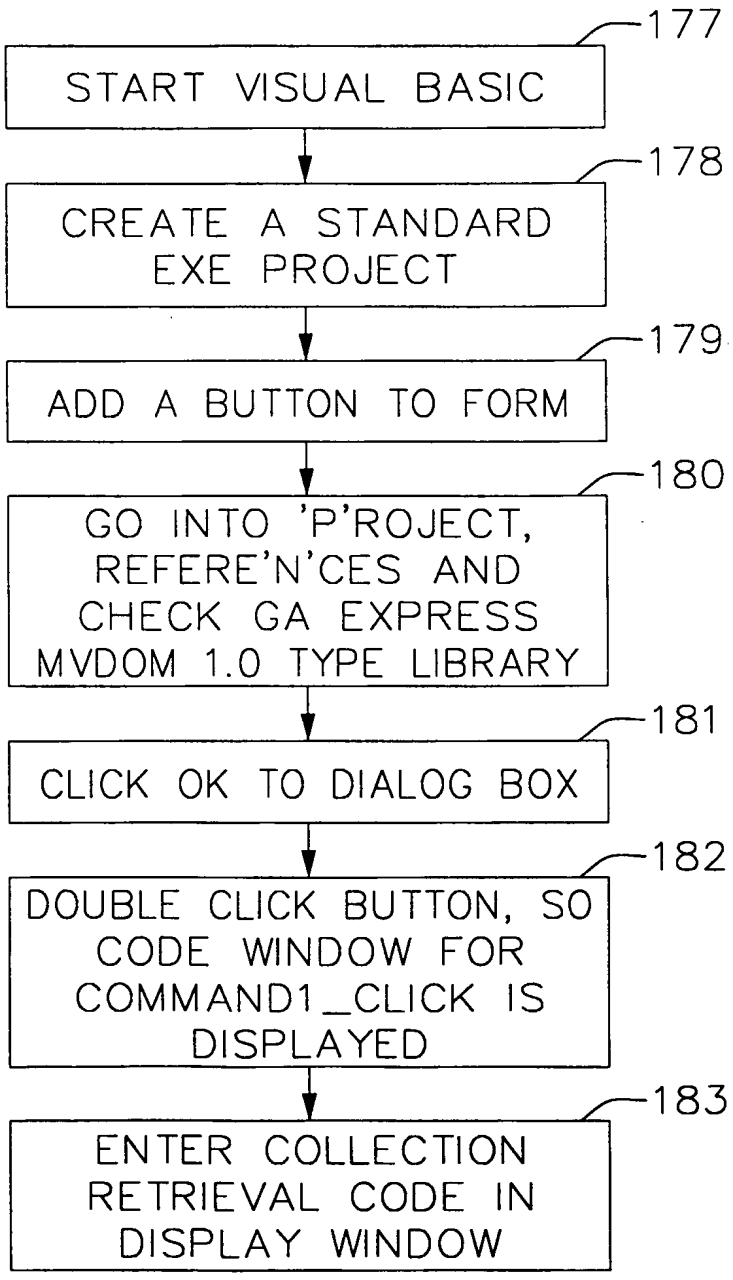
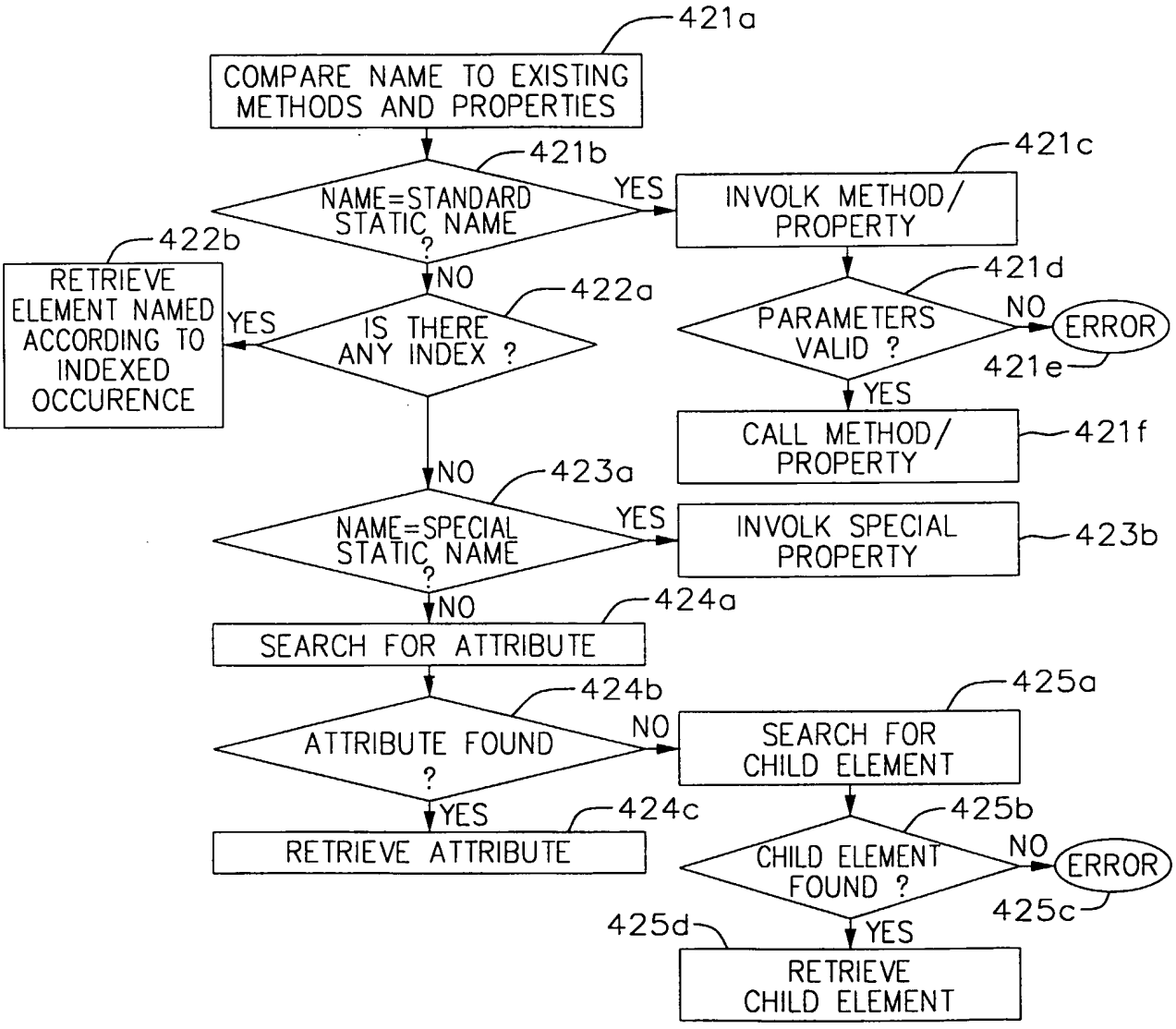

177
START VISUAL BASIC
178
CREATE A STANDARD EXE PROJECT
179
ADD A BUTTON TO FORM
180
GO INTO 'P'ROJECT, REFERE'N'CES AND CHECK GA EXPRESS MVDOM 1.0 TYPE LIBRARY
181
CLICK OK TO DIALOG BOX
182
DOUBLE CLICK BUTTON, SO CODE WINDOW FOR COMMAND1_CLICK IS DISPLAYED
183
ENTER COLLECTION RETRIEVAL CODE IN DISPLAY WINDOW

FIG.28

```
400—PRIVATE SUB COMMAND1_CLICK()
        401—DIM OBJMVD AS NEW GAXLIB.MVDOM
        402—OBJMVD.CREATEROOT ("MVDATE").APPEND("RECORD").
403—APPENDATTRIBUTE("ID")="ABC"
    404—OBJMVD.MVDATA.APPEND("RECORD").
405—APPENDATTRIBUTE("ID")="DEF"
    406—OBJMVD.MVDATA.APPEND("RECORD")
407—APPENDATTRIBUTE("ID") = "GHI"
    408—OBJMVD.MVDATA.APPEND("FOO") = "BAR"
    409—FOR EACH RECITEM IN OBJMVD.MVDATA.CHILDNODES()
        410—{MSGBOX RECITEM.NAME, , _
             "FOR EACH WITH CHILDNODES()"
     411—NEXT
    412—{FOR EACH RECITEM IN
         OBJMVD.MVDATA.CHILDNODES("RECORD")
        413—{MSGBOX RECITEM.NAME, , _
             "FOR EACH WITH CHILDNODES(""RECORD"")"
    414—NEXT
    415—DIM OBJMVCOLL AS NEW GAXLIB.MVCOLLECTION
    416—SET OBJMVCOLL = OBJMVD.MVDATA.CHILDNODES()
    417—FOR i = 1 TO OBJMVCOLL.COUNT
        418—{MSGBOX CSTR(i) & "=" & OBJMVCOLL.ITEM(i).
             NAME, , _ "LOOPING THROUGH USING ITEMS(n)"
    419—NEXT
419—END SUB
```

FIG.29


421a
COMPARE NAME TO EXISTING METHODS AND PROPERTIES
421b
NAME=STANDARD STATIC NAME ?
YES
421c
INVOLK METHOD/ PROPERTY
NO
422b
RETRIEVE ELEMENT NAMED ACCORDING TO INDEXED OCCURENCE
YES
422a
IS THERE ANY INDEX ?
421d
PARAMETERS VALID ?
NO
ERROR
421e
YES
CALL METHOD/ PROPERTY
421f
NO
423a
NAME=SPECIAL STATIC NAME ?
YES
INVOLK SPECIAL PROPERTY
423b
NO
424a
SEARCH FOR ATTRIBUTE
424b
ATTRIBUTE FOUND ?
NO
425a
SEARCH FOR CHILD ELEMENT
YES
424c
RETRIEVE ATTRIBUTE
425b
CHILD ELEMENT FOUND ?
NO
ERROR
425c
425d
YES
RETRIEVE CHILD ELEMENT

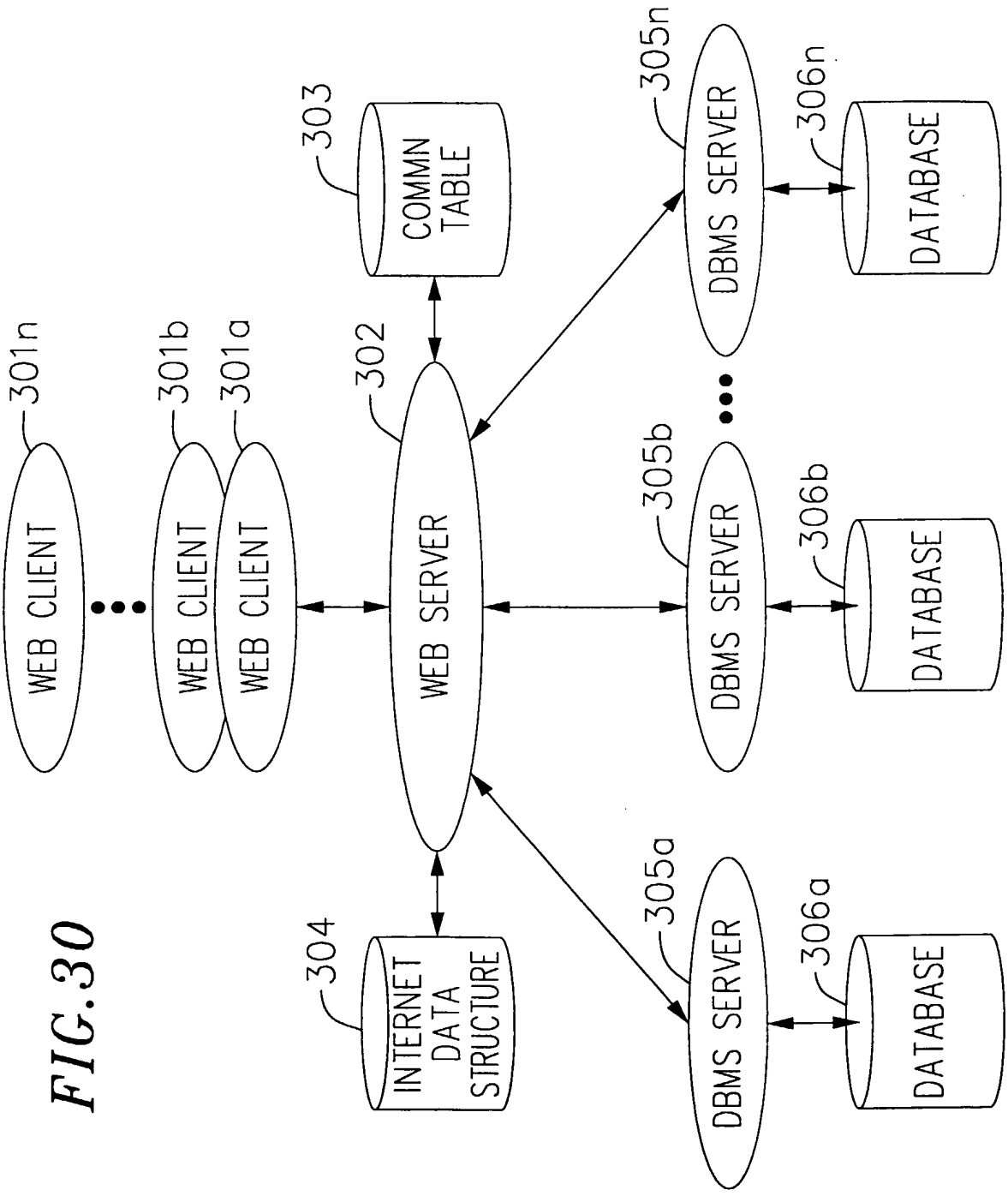

WEB CLIENT
WEB CLIENT
WEB CLIENT
301n
301b
301a
WEB SERVER
302
COMMN TABLE
303
INTERNET DATA STRUCTURE
304
DBMS SERVER
DBMS SERVER
DBMS SERVER
305n
305b
305a
DATABASE
DATABASE
DATABASE
306n
306b
306a

FIG. 30

FIG.31

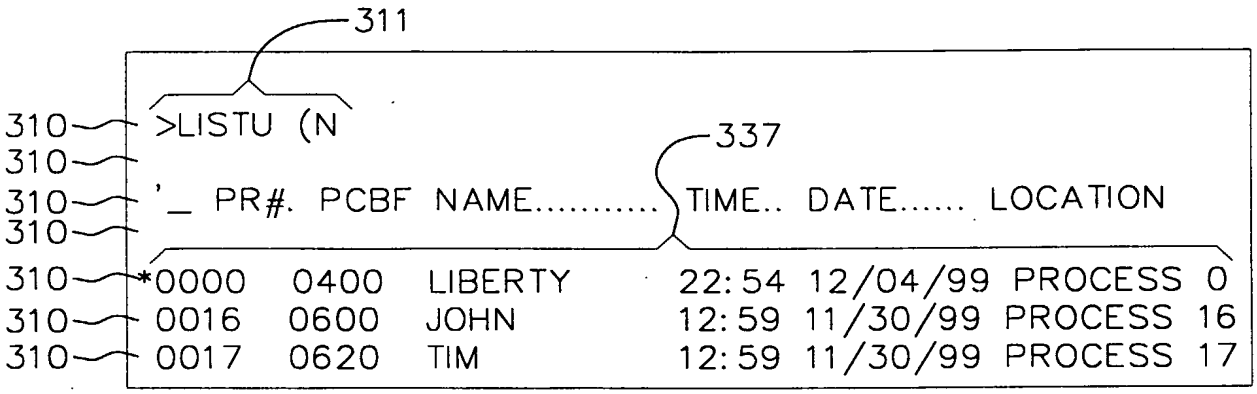


FIG.32

```
  350—<%@Language=VBScript %>
351-1—<%
  352—set obj=Server.CreateObject("BizObject.BizAdapter")
  353—obj.DataSource="dbms.myserver.com"
  354—obj.UserId="ORDERENTRY"
  355—obj.Password="MYPASSWORD"
  356—x=obj.readXML("EXE TCL 'LISTU (N'")
  357—Response.ContentType="text/xml"
  358—Response.Write(xml)
351-2—%>
```

FIG.33

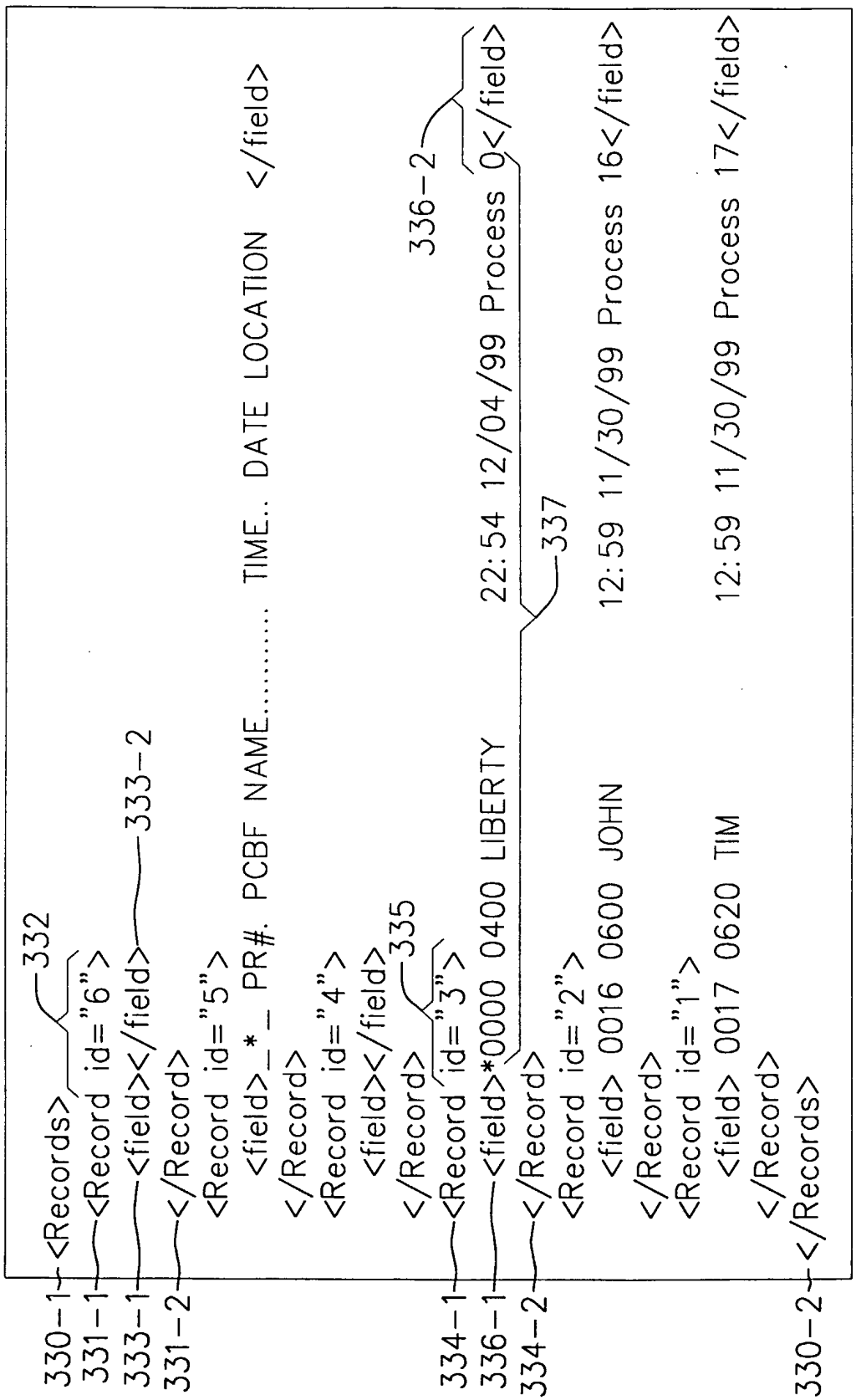

330-1
331-1
333-1
331-2
334-1
336-1
334-2
330-2
<Records>
<Record id="6">
<field></field>
</Record>
<Record id="5">
<field>_*_ PR#. PCBF NAME..... TIME.. DATE LOCATION </field>
</Record>
<Record id="4">
<field></field>
</Record>
<Record id="3">
<field>*0000 0400 LIBERTY 22:54 12/04/99 Process 0</field>
</Record>
<Record id="2">
<field> 0016 0600 JOHN 12:59 11/30/99 Process 16</field>
</Record>
<Record id="1">
<field> 0017 0620 TIM 12:59 11/30/99 Process 17</field>
</Record>
</Records>
332
333-2
335
336-2
337

FIG.34

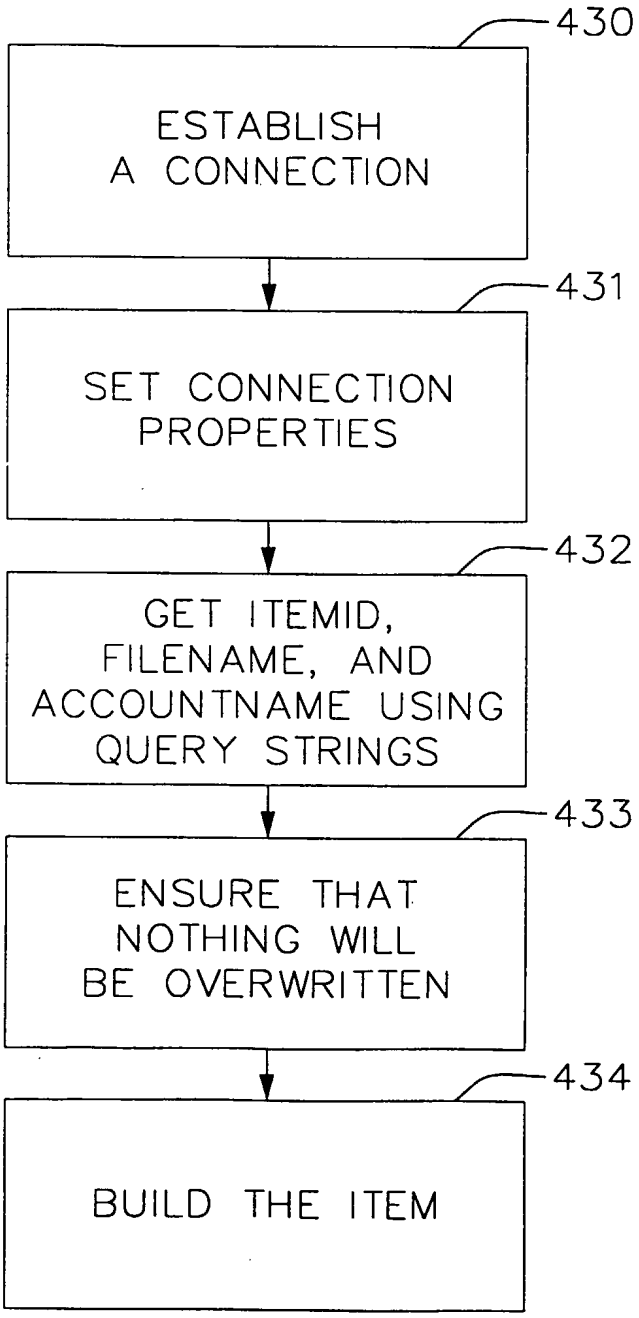

430
ESTABLISH A CONNECTION
431
SET CONNECTION PROPERTIES
432
GET ITEMID, FILENAME, AND ACCOUNTNAME USING QUERY STRINGS
433
ENSURE THAT NOTHING WILL BE OVERWRITTEN
434
BUILD THE ITEM

FIG.35

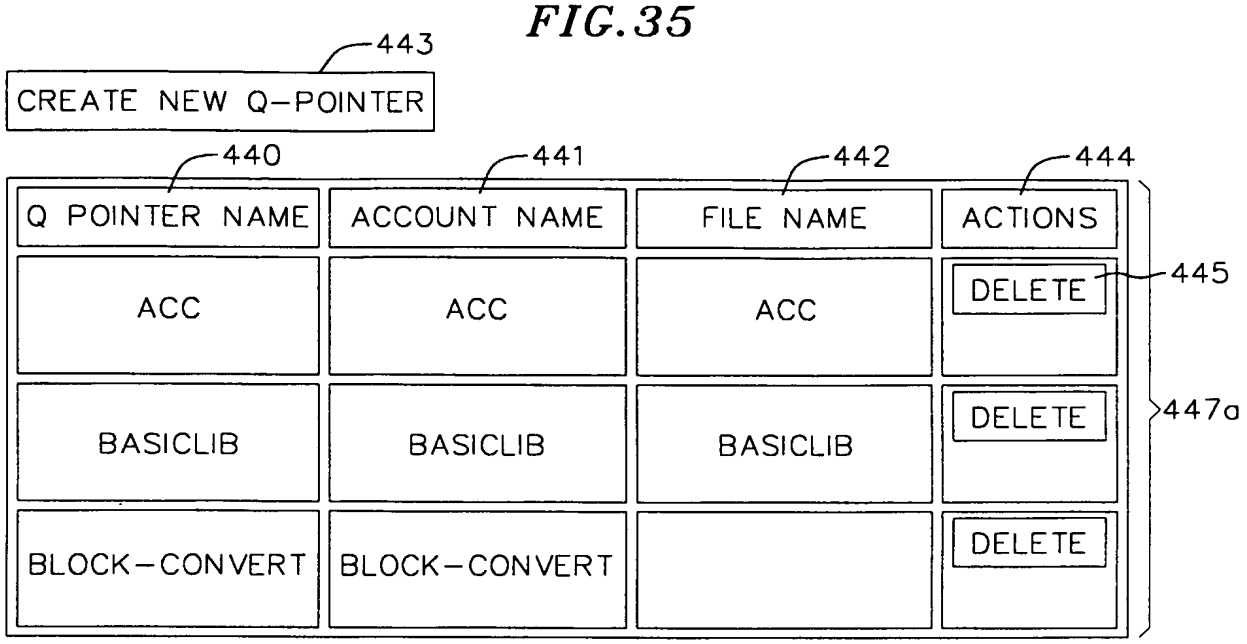

443
CREATE NEW Q-POINTER
440
441
442
444
Q POINTER NAME
ACCOUNT NAME
FILE NAME
ACTIONS
ACC
ACC
ACC
DELETE
445
BASICLIB
BASICLIB
BASICLIB
DELETE
447a
BLOCK-CONVERT
BLOCK-CONVERT
DELETE

FIG.36

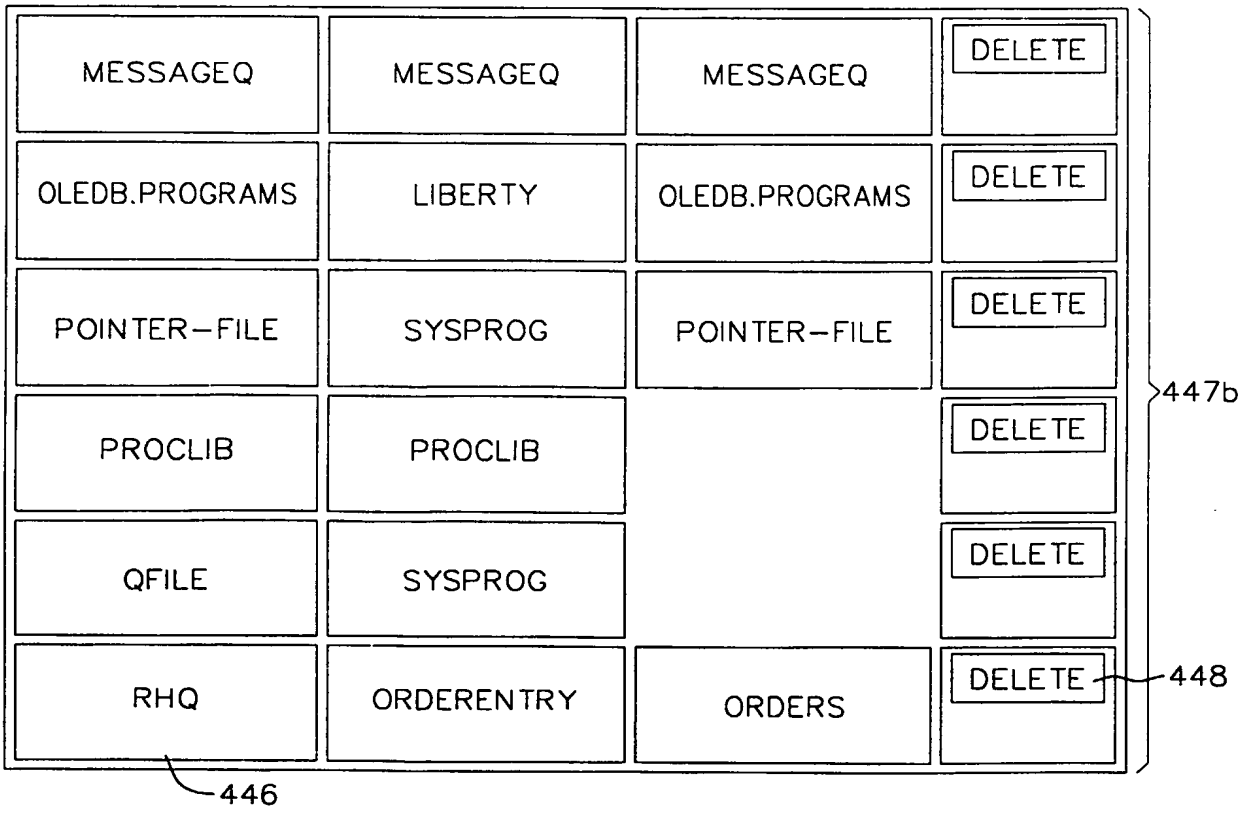

MESSAGEQ
MESSAGEQ
MESSAGEQ
DELETE
OLEDB.PROGRAMS
LIBERTY
OLEDB.PROGRAMS
DELETE
POINTER-FILE
SYSPROG
POINTER-FILE
DELETE
447b
PROCLIB
PROCLIB
DELETE
QFILE
SYSPROG
DELETE
RHQ
ORDERENTRY
ORDERS
DELETE
448
446

FIG.37

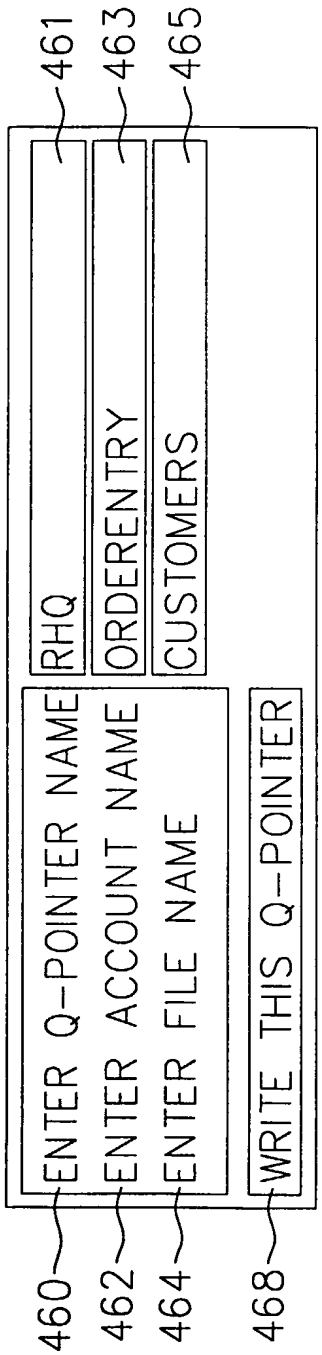

461
463
465
RHQ
ORDERENTRY
CUSTOMERS
ENTER Q-POINTER NAME
ENTER ACCOUNT NAME
ENTER FILE NAME
WRITE THIS Q-POINTER
460
462
464
468

FIG.38

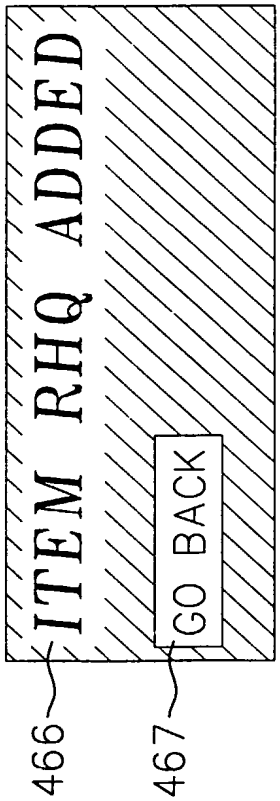

ITEM RHQ ADDED
GO BACK
466
467

FIG.39

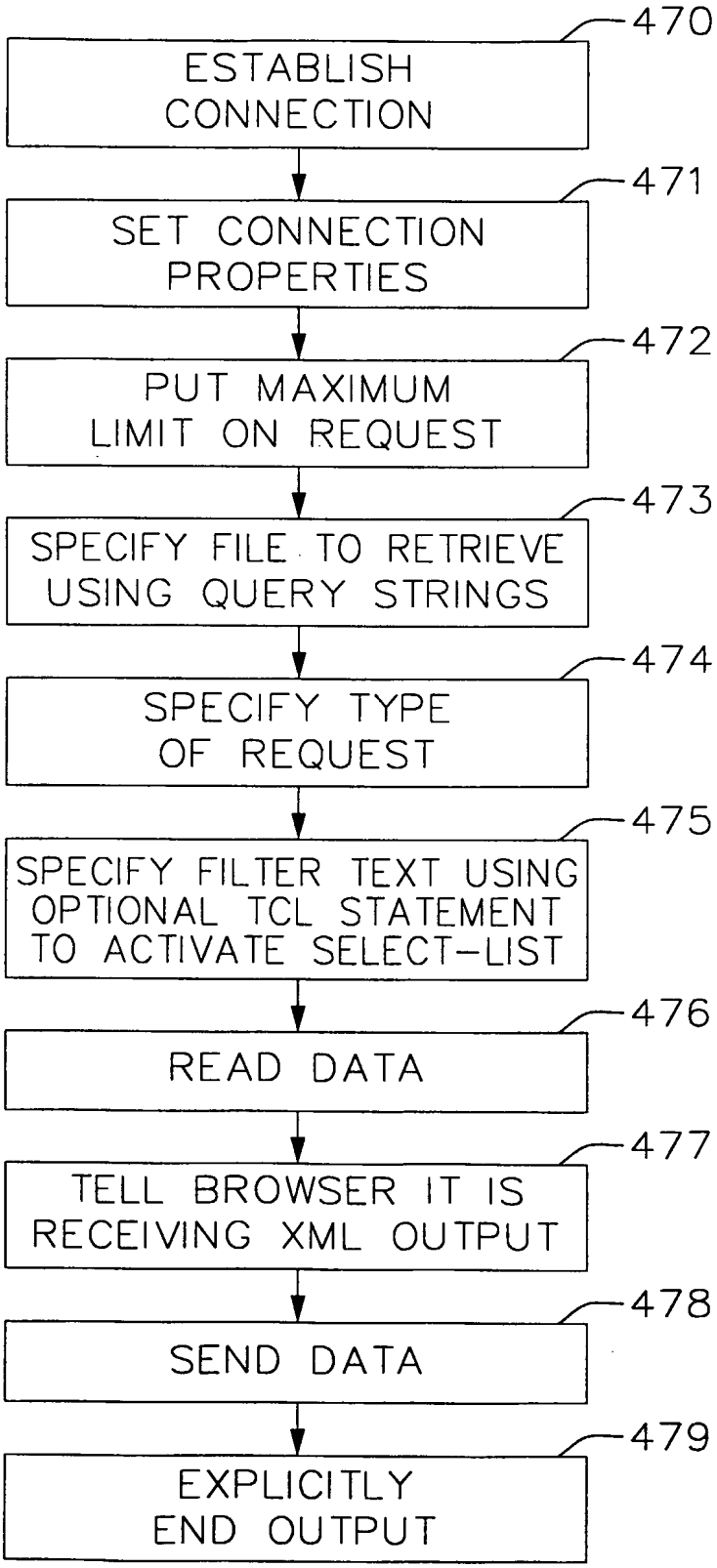

470
ESTABLISH CONNECTION
471
SET CONNECTION PROPERTIES
472
PUT MAXIMUM LIMIT ON REQUEST
473
SPECIFY FILE TO RETRIEVE USING QUERY STRINGS
474
SPECIFY TYPE OF REQUEST
475
SPECIFY FILTER TEXT USING OPTIONAL TCL STATEMENT TO ACTIVATE SELECT-LIST
476
READ DATA
477
TELL BROWSER IT IS RECEIVING XML OUTPUT
478
SEND DATA
479
EXPLICITLY END OUTPUT

FIG. 40

480
481
ADDRESS
http//rhacer.libertyodbc.com/Demo/GetFile.asp?FileName=RHQ&FilterText=SELECT
RADIO
PLAY
RADIO STATIONS
DiscJockey.Com Christmas Fur
-<MVData xml:space="preserve">
-<record id="1">
<field>R&B Auto Repair</field>
</record>
</MVData>
485

FIG. 41

482
483
http//rhacer.libertyodbc.com/Demo/GetFile.asp?
FileName=RHQ&FilterText=SELECT%20RHQ%20'1'
485

FIG. 42

```
486 maxItems=100
487 FileName=Request.QueryString("FileName")
488 IF FileName = "" Then
        FileName=Request.Form.Item("FileName")
489 End if
490 FilterText=Request.QueryString("FilterText")
491 if FilterText = "" then
        FilterText=Request.Form.Item("FilterText")
492 end if
```

FIG. 43

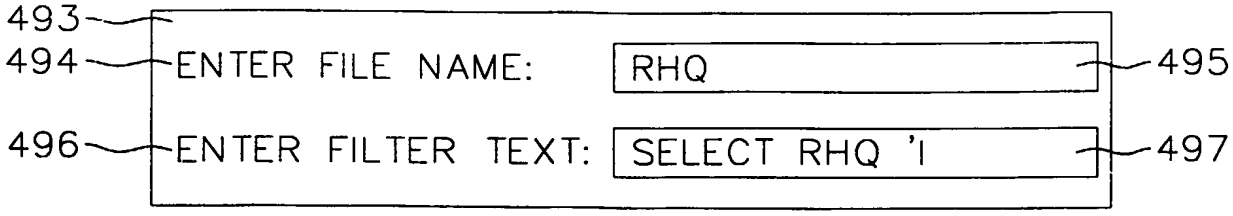
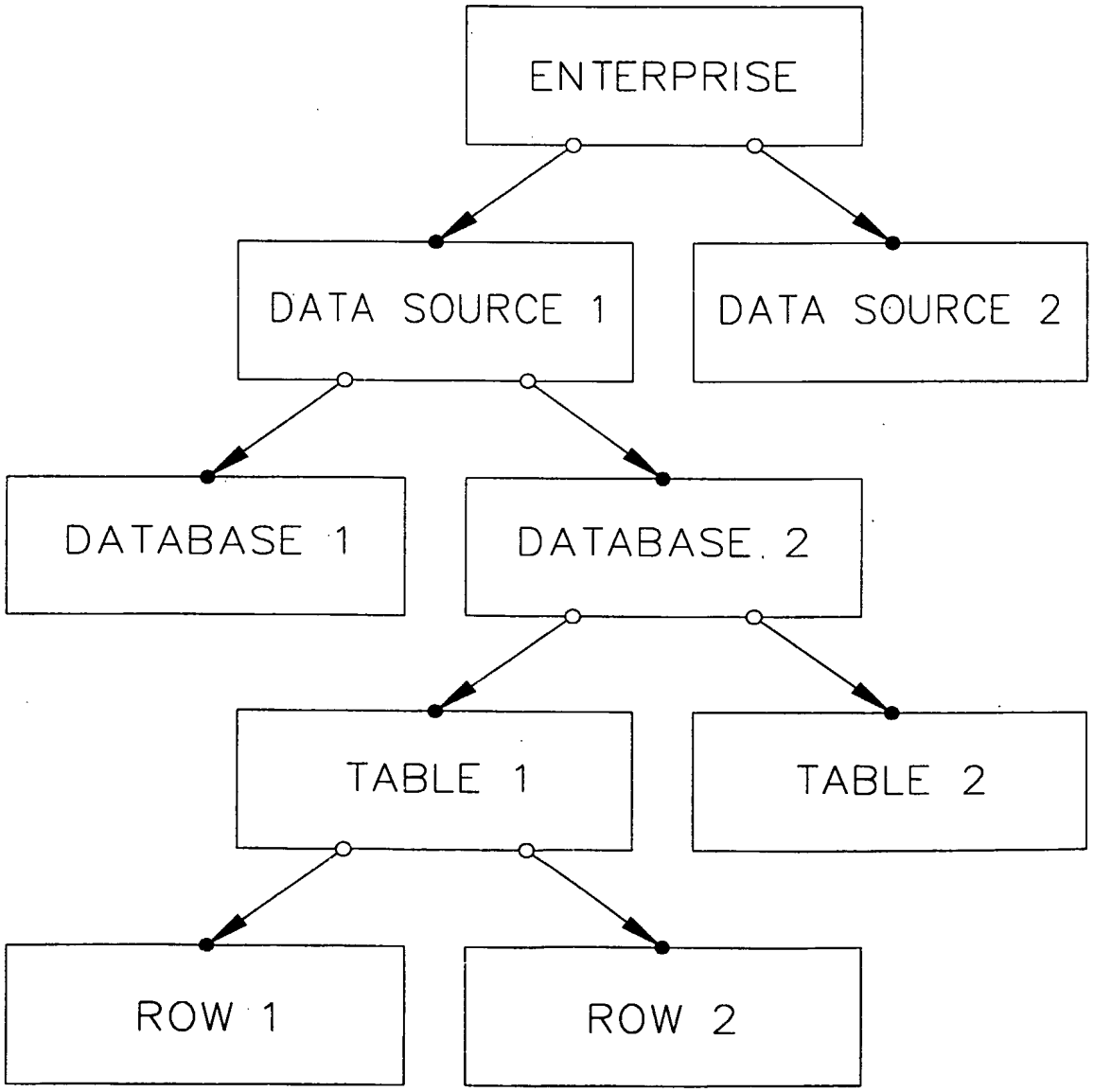


FIG. 44

```
500~xml=obj.ReadXML(FileName,Filtertext,maxItems)

501~Response.ContentType="text/xml"

502~Response.Write(xml)

503~Response.end
```

FIG. 45

FIG. 46


ENTERPRISE
DATA SOURCE 1
DATA SOURCE 2
DATABASE 1
DATABASE 2
TABLE 1
TABLE 2
ROW 1
ROW 2